Anwendung und Untersuchung von Path-Packing in der Lagerlogistik

Alexander Büchel

Publisher: Dean Prof. Dr. Wolfgang Heiden

University of Applied Sciences Bonn-Rhein-Sieg,
Department of Computer Science

Sankt Augustin, Germany

September 2016

Technical Report 02-2016

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

ISSN 1869-5272

Digital Object Identifier doi:10.18418/978-3-96043-033-9
DOI-Resolver http://dx.doi.org/

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

EHRHARDT+PARTNER

Abschlussarbeit

im Studiengang Master of Science in Computer Science

Anwendung und Untersuchung von Path-Packing in der Lagerlogistik

von
Alexander Büchel

Erstbetreuer:	Prof. Dr. Peter Becker Hochschule Bonn-Rhein-Sieg Fachbereich Informatik
Zweitbetreuer:	Prof. Dr. Kurt-Ulrich Witt Hochschule Bonn-Rhein-Sieg Fachbereich Informatik
Ext. Betreuer:	Dipl. Ing. (FH) Jens Heinrich Ehrhardt + Partner GmbH & Co. KG
Abgabedatum:	20.07.2015

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	**Bedeutung**
BFS	Breadth First Search
BLB	Best Local Bound
BPH	Best Projection Heuristic
CLQ	Clique Cover Cut
COV	Mixed Cover Cut
CSV	Comma Seperated Values
DAG	Directed Acyclic Graph
DFS	Depth First Search
DTH	Heuristic By Driebeck & Tomlin
FFV	First Fractional Variable
gdw.	genau dann wenn
GlpK	Gnu Linear Programming Toolkit
GMI	Gomory's Mixed Integer Cut
LFV	Last Fractional Variable
LP	Lineares Programm
MFV	Most Fractional Variable
MIP	Mixed Integer Programming
MIR	Mixed Integer Rounding
PCH	Hybrid Pseudo-Cost Heuristic
UML	Unified Modeling Language
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Der Arbeitsablauf innerhalb eines Logistiklagers ist ein Zusammenspiel unterschiedlichster Akteure und Faktoren. In diesem Zusammenspiel besitzen Auftragsgeber, deren Produktanfragen zusammengestellt werden müssen, die Kommissionierer und deren Transport- und Fortbewegungsmöglichkeiten innerhalb des Lagers, aber auch der Typ eines Lagers eine wichtige Rolle. Unter wissenschaftlichen Gesichtspunkten können die Mengen an Aufträgen, die Anzahl an Mitarbeitern und die unterschiedlichsten Eigenschaften von Produkten zu sehr komplexen Abläufen führen, die reibungsfrei funktionieren müssen.

1.1 Motivation

Über viele Stunden täglich müssen die genannten Faktoren bestmöglich aufeinander abgestimmt sein, ohne dass der Arbeitsablauf unterbrochen wird. Die Abschlussarbeit wird in Kooperation mit dem Unternehmen „Ehrhardt + Partner" mit Hauptsitz in Boppard-Buchholz angefertigt. „Ehrhardt + Partner" bietet Lösungen in der Warehouse-Logistik an. Der reibungslose Arbeitsablauf wird durch das Lagerführungssystem LFS gewährleistet. (vgl. Ehrhardt+Partner 2015b; Ehrhardt+Partner 2015a, S. 4ff.)

Im Allgemeinen existieren zwei verschiedene Typen von Lagern, die sich in der Art der Bereitstellung der Ware unterscheiden: das „Mann-zur-Ware-Lager" und das „Ware-zum-Mann-Lager". Im erst genannten Lagertyp befindet sich die Ware an statischen Positionen innerhalb des Lagers; der Kommissionierer muss die verlangten Lagerplätze nacheinander aufsuchen und die jeweilige Ware aus den Regalen nehmen. Im Vergleich dazu ist das „Ware-zum-Mann"-Prinzip dynamischer strukturiert, da die Ware aus ihren Lagerpositionen automatisiert zu den Kommissionierplätzen transportiert wird. Die statische Komponente ist somit der Mitarbeiter, der von einem festen Arbeitsplatz aus die zu ihm transportierten Waren in Empfang nehmen muss (vgl. Heinrich 2014, S. 397ff.).

Diese Abschlussarbeit beschäftigt sich mit Lagern, die zu dem zuletzt genannten Lagertyp gehören. Hierbei werden Lager betrachtet, bei denen Behälter, meistens Kisten, über ein Transportsystem bewegt werden. Durch gegebene Auftragslisten werden bestimmte Mengen an Waren einem Behälter zugeordnet, sodass diese Behälter die zugeordneten Warenpositionen im Lager anfahren müssen. Diese Waren befinden sich an den statischen Arbeitsplätzen, die auch „Bahnhöfe" genannt werden. An diesen stehen Mitarbeiter, die die bereitgestellten Warenmengen aus den Kommissionierregalen in den jeweiligen Behälter geben und diesen erneut auf das Transportnetz zurückführen.

Eine hohe Auslastung innerhalb dieses Transportsystems ist das zu erreichende Ziel und somit auch die Motivation für die Fragestellung dieser Abschlussarbeit. Ein effizienter Transportablauf liegt genau dann vor, wenn Kisten über das Transportnetz bewegt werden, ohne dass Staus oder Blockaden entstehen. Die Fragestellung nach einer hohen Auslastung wird in dieser Abschlussarbeit auf mathematischer Ebene untersucht.

1.2 Problemstellung

Um dieses praxisnahe Problem näher untersuchen zu können, muss ein mathematisches Modell geformt werden. Dieses stellt die Basis für die Anwendung verschiedener Algorithmen und Heuristiken dar.

Ein solches Transportsystem kann durch die Graphentheorie als Flussnetz abgebildet werden. Behälter werden über die Quelle in dieses System hineingegeben und verlassen das System über die Senke. Die Transportbahnen werden durch Kanten repräsentiert, die durch eine bestimmte Kapazität beschränkt sind. Über Zwischenstationen, auch Hubs genannt, können diese Behälter über unterschiedliche Transportbahnen

geleitet werden. Diese sind folglich die Knoten des Flussnetzes. Der minimale Schnitt liefert die obere Schranke an gleichzeitig zu transportierenden Behältern.

Jedem Behälter wird ein bestimmter Auftrag zugeordnet, der wiederum eine Warenmenge enthält, die in diesen Behälter hineingepackt werden muss. Daraus kann der Weg abgeleitet werden, der von einem Behälter durch das Flussnetz genommen werden muss. Dieser Weg, angefangen von der Quelle, über verschiedene Zwischenstationen, bis hin zur Senke, wird auch als Pfad (engl. path) bezeichnet. Der Behälter kann folglich seinen Weg nicht frei wählen, sondern besitzt aufgrund der anzufahrenden Bahnhöfe einen bestimmten Weg. Durch das Transportieren einer Kiste und dem daraus folgenden Besetzen einer Kante steigen die aktuellen Flusswerte der Kanten innerhalb dieses Flussnetzes. Diese Flusswerte sind nach oben beschränkt durch den Wert der jeweiligen Kantenkapazität.

Durch unterschiedliche Eigenschaften von Aufträgen, wie bspw. die Abfahrtzeiten von Lkws oder verschiedene Kundenklassen, müssen bestimmte Behälter prioritär abgearbeitet werden. Diese prioritäre Abarbeitung wird durch einen Nutzwert repräsentiert. Durch die Modellierung des Transportnetzes, die Definition der Pfadmengen und die zugewiesenen Nutzwerte kann nun die zentrale Fragestellung der Arbeit formuliert werden: Welche Behälter können gleichzeitig durch das Transportnetz transportiert werden, ohne dass erstens eine Kantenkapazität überschritten wird und zweitens ein möglichst hoher Gesamtnutzwert erreicht wird? Dieses Auswahlverfahren wird folglich als „Path-Packing" bezeichnet.

1.3 Zielsetzung der Arbeit

In dieser Abschlussarbeit ist die primäre Aufgabe, verschiedene Heuristiken und Algorithmen auf Basis des graphentheoretischen Modells zu definieren, zu beschreiben und zu analysieren. Insbesondere soll dabei die Tauglichkeit des Algorithmus bzw. der Heuristik in Bezug auf die jeweilige Laufzeit untersucht werden. Die benötigte Laufzeit ist ein wichtiges Indiz für die Anwendbarkeit in der Praxis.

Da das Path-Packing und auch verwandte mathematische Probleme NP-vollständig sind und somit für optimale Lösungen eine exponentielle Laufzeit besitzen, sollen hierbei nicht nur optimale, sondern auch approximative Lösungen betrachtet werden. Heuristiken besitzen meist eine effizientere Laufzeit. Der Nachteil kann allerdings darin liegen, dass eine Näherungslösung nicht nah genug an der optimalen Lösung liegt und folglich keine zufriedenstellende Lösung darlegt. Der zweite Untersuchungsaspekt ist daher die qualitative Leistung, d.h. die Differenz zwischen der Näherungslösung und der Optimallösung, die von der jeweiligen Heuristik erlangt werden kann.

Für die Evaluation dieser Algorithmen sind geeignete Datenmengen nötig. Daher ist die Generierung von Testdaten ebenfalls eine wichtige Aufgabe. Dafür müssen nicht nur Flussnetze, sondern auch die Pfadmengen mit verschiedenen Eigenschaften randomisiert erstellt werden.

1.4 Gliederung der Arbeit

Diese Abschlussarbeit wird in fünf Hauptkapitel gegliedert. Einleitend werden dem Leser die Problemstellung und Zielsetzung dieser Arbeit näher gebracht (Kapitel 1). Kapitel 2 beschäftigt sich mit den notwendigen theoretischen Grundlagen. Dies betrifft sowohl die Grundlagen der Graphentheorie, der linearen Optimierung als auch verwandten Optimierungsproblemen, die Ähnlichkeiten mit der Fragestellung des Path-Packings aufweisen. Auf Basis dieser Grundlagen werden vier algorithmische Ansätze vorgestellt und theoretisch analysiert (Kapitel 3). Die Generierung von notwendigen Testdaten, die Evaluation der Ergebnisse und eine Bewertung des definierten Modells findet in Kapitel 4 statt. Die Abschlussarbeit schließt mit einer Zusammenfassung ab und gibt einen Ausblick über zukünftige Ausbesserungen hinsichtlich der hier durchgeführten Path-Packing-Modellierung (Kapitel 5).

2 Theoretische Grundlagen

In diesem Kapitel werden die Grundlagen zur mathematischen Modellierung der Transportnetze, der zu generierenden Pfadmengen sowie der verwendeten Algorithmen gelegt. Die Definitionen der graphentheoretischen Grundlagen, die Beschreibung der allgemeinen Optimierungsproblematik sowie das Lösen dieser Optimierungsprobleme sind von wesentlicher Bedeutung. Zusätzlich werden konkrete Optimierungsprobleme angesprochen, die auf theoretischer Ebene einen nahen Bezug zum Path-Packing aufweisen.

2.1 Notationen innerhalb der Arbeit

Da in literarischen Quellen zum Teil unterschiedliche Notationen für graphentheoretische Ausdrücke existieren, werden in diesem Abschnitt die Notationen mit ihren Bedeutungen festgelegt, wie sie in dieser Arbeit verwendet werden. Tabelle 2.1 fasst diese zusammen:

Notation	**Bedeutung**
$G(V, E)$	Ein allgemeiner Graph G mit einer Knotenmenge V und einer Kantenmenge E.
$F(V, E)$	Ein Flussnetz F mit einer Knotenmenge V und einer Kantenmenge E.
$n = \|V\|$	Anzahl der Knoten in der Knotenmenge V.
$m = \|E\|$	Anzahl der Kanten in der Kantenmenge E.
$c(e)$ mit $e \in E$	Die Kapazität c einer Kante e aus E.
$p = \|P\|$	Anzahl der Pfade in der Pfadmenge P.
$p_i =< (v_1, v_2), \ldots, (v_{n-1}, v_n) >$	Ein Pfad p_i bestehend aus einer Folge von Kanten. Dabei gilt, dass der erste Knoten v_1 die Quelle ist und der letzte Knoten v_n die Senke ist.
$d^-(v)$	Eingangsgrad von v - Die Anzahl der Kanten, die in einen Knoten v hineingehen.
$d^+(v)$	Ausgangsgrad von v - Die Anzahl der Kanten, die aus einem Knoten v herausgehen.
q bzw. v_1	Die Quelle eines Flussnetzes wird mit q abgekürzt. Im Zusammenhang mit der algorithmischen Erstellung der Flussnetze ist der erste erstellte Knoten die Quelle. Somit gilt, dass die Quelle mit v_1 bezeichnet wird.
s bzw. v_n	Die Senke eines Flussnetzes wird im Allgemeinen mit s abgekürzt. Im Zusammenhang mit der algorithmischen Erstellung der Flussnetze ist der letzte erstellte Knoten immer die Senke. Ein Flussnetz mit n Knoten, besitzt daher die Senke v_n.

Tab. 2.1: Verwendete Notationen

2.2 Graphentheoretische Grundlagen

Die mathematische Darstellung eines Transportnetzes kann durch ein Flussnetz realisiert werden. Dieser Abschnitt beschäftigt sich mit eben dieser Aufgabe und definiert dabei die Lagerstruktur, basierend auf graphentheoretischen Grundlagen und Eigenschaften.

2.2.1 Definitionen innerhalb eines Flussnetzwerkes

Bevor eine Untersuchung von Algorithmen in Bezug auf das Path-Packing stattfinden kann, müssen zunächst die Lagerstrukturen durch ein geeignetes Modell abgebildet werden. Die folgenden Definitionen von relevanten Fachbegriffen basieren dabei auf (Turau 2009, S. 173ff.).

Ein Logistiklager kann mit Hilfe der Graphentheorie durch ein *Flussnetzwerk* – auch *Flussnetz* genannt – beschrieben werden. Dieses Flussnetz ist gleichbedeutend mit einem gerichteten Graphen $G(V, E)$, dessen Kanten eine nichtnegative Kapazität $c(e)$ zugewiesen bekommen. Weiterhin gilt, dass zusätzlich jedes Flussnetz eine Quelle q und eine Senke s besitzen muss, die als Knoten in der Knotenmenge existieren. Die Quelle q zeichnet sich dadurch aus, dass sie keine eingehenden Kanten besitzt; die Senke s dagegen keine ausgehenden Kanten. Dies wird wie folgt notiert: $d^-(q) = d^+(s) = 0$.

Weiterhin ist der Begriff *Fluss* von wichtiger Bedeutung. Ein Fluss ist eine Funktion f, die jeder Kante $e \in E$ eines Graphen eine Zahl zuordnet, deren Wert zwischen 0 und der Kapazität c der jeweiligen Kante liegt: $0 \leq f(e) \leq c(e)$. Außerdem gilt für jeden Knoten $v \in V$, dass die Summe der Flusswerte, die in diesen Knoten hineingehen, gleich der Summe der Flusswerte ist, die aus ihm herausgehen:

$$\forall v \in V \setminus \{q, s\} : \sum_{e=(w,v)\in E} f(e) = \sum_{e=(v,w)\in E} f(e)$$

Diese Bedingung gilt für alle Knoten mit Ausnahme der Quelle q und der Senke s. Ebenfalls gilt, dass der gesamte Fluss f innerhalb dieses Flussnetzes aus der Quelle kommen muss und verlustfrei in die Senke mündet:

$$\sum_{e=(q,v)\in E} f(e) = |f| = \sum_{e=(v,s)\in E} f(e)$$

2.2.2 MaxFlow-MinCut-Theorem

Eine wichtige Frage bei der Untersuchung von Flüssen ist die Frage nach dem größten Fluss, auch *maximaler Fluss* genannt. Um diesen zu ermitteln, wird zunächst ein *Schnitt* in einem Flussnetzwerk definiert. Ein Schnitt ist eine Aufteilung der Knotenmenge in zwei disjunkte Knotenmengen X und $\overline{X}$ des Graphen mit $q \in X$ und $s \in \overline{X}$. Die Summe der Kantenkapazitäten, die von X nach $\overline{X}$ verlaufen, wird als Kapazität des Schnitts bezeichnet. Mathematisch wird dies wie folgt definiert:

$$c(X, \overline{X}) = \sum_{e=(v,w)\ v\in X, w\in\overline{X}} c(e)$$

Analog dazu wird der Fluss des Schnittes als Summe der Flusswerte dieser Kanten von X nach $\overline{X}$ definiert:

$$f(X, \overline{X}) = \sum_{e=(v,w)\ v\in X, w\in\overline{X}} f(e)$$

Für jeden Schnitt $(X, \overline{X})$ eines Graphen gelten somit zwei Bedingungen (vgl. Turau 2009, S. 176):

1. $|f| = f(X, \overline{X}) - f(\overline{X}, X)$
2. $|f| \leq \min\{c(X, \overline{X}) | (X, \overline{X}) \text{ ist Schnitt von G}\}$

Da bei einem gültigen Fluss die Bedingung $0 \leq f(e) \leq c(e)$ erfüllt sein muss, gilt weiterhin auch:

$$|f| = f(X, \overline{X}) - f(\overline{X}, X) \leq f(X, \overline{X}) \leq c(X, \overline{X})$$

Daraus folgt, dass der minimale Schnitt somit eine obere Grenze für den Wert des maximalen Flusses bildet. Nach dem Satz von Ford und Fulkerson (vgl. Turau 2009, S. 179) ist der Wert des maximalen Flusses identisch mit dem des minimalen Schnittes.

2.2.3 Pfad-Definition

Ein *Pfad* (engl. path) stellt den Weg einer Kiste dar und somit eine Folge von Kanten innerhalb dieses Flussnetzes. Dieser Weg beginnt grundsätzlich in der Quelle q und endet in der Senke s. Die Notation für einen Pfad sei die folgende: $p_i =< (v_1, v_2) \dots (v_{n-1}, v_n) >$ (vgl. Syslo u. a. 2007, S. 226). Bei allgemeinen gerichteten Graphen kann ein Pfad allerdings auch Zyklen enthalten. Ein Zykel ist in diesem Zusammenhang so definiert, dass ein Pfad zyklisch ist, gdw. eine Kante doppelt in ihm vorkommt. Eine Kiste belegt beim Transport über ihre Kanten jeweils eine Kapazitätseinheit dieser Kante. Die Pfade aller Kisten, die gleichzeitig über das Transportnetz geschickt werden, bilden somit den Fluss des Netzes.

Somit wird der Zusammenhang zwischen dem Transport der Kisten und den graphentheoretischen Grundlagen klar: Der minimale Schnitt eines Graphen ist der Wert des maximalen Flusses und folglich die maximale Anzahl an Kisten, die gleichzeitig transportiert werden können. Er bildet somit eine obere Schranke für diese Anzahl an Kisten. Um von der Quelle zur Senke zu gelangen, muss jede Kiste mindestens einmal den Engpass des Flussnetzes, der die Grundlage für den minimalen Schnitt ist, durchlaufen.

Für ein geeignetes Path-Packing-Modell muss jedem Pfad noch ein Nutzwert zugewiesen werden. Dieser repräsentiert die Priorität für das Abarbeiten, wie bspw. Lkw-Abfahrtzeiten oder Kundenklassen. Für diese Abschlussarbeit genügt es, den Pfaden einen randomisierten numerischen Wert zuzuweisen. Eine Herleitung der Berechnung wird nicht genauer betrachtet; allerdings wird in Kapitel 5.2 ein Ausblick über eine Herleitung gegeben.

2.2.4 Beispiel eines Flussnetzes

Um den Zusammenhang dieser Definitionen besser zu verdeutlichen, sollen diese anhand des folgenden Schaubildes graphisch dargestellt werden. Abbildung 2.1 zeigt ein Flussnetz $F(V, E)$ mit

$$V = \{q, a, b, c, d, s\}$$

und

$$E = \{(q, a), (q, c), (q, d), (d, a), (d, b), (a, b), (b, c), (c, s), (a, s)\}.$$

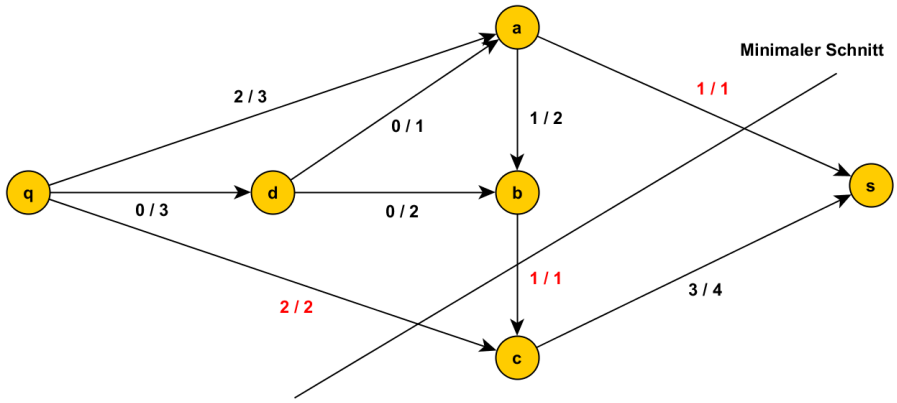

Abb. 2.1: Beispiel eines Flussnetzes

Der Minimale Schnitt ist definiert durch die beiden disjunkten Knotenmengen $X = \{q, a, b, d\}$ und $\overline{X} = \{c, s\}$. Der Wert des minimalen Schnitts ist somit die Summe der Kapazitäten der Kanten $(a, s), (b, c)$ und (q, c), die aus X in $\overline{X}$ gehen und besitzt in diesem Beispiel den Wert 2 + 1 + 1 = 4. Die Folgerung ist, dass der maximale Fluss ebenfalls den Wert 4 hat.

Übertragen in das „Path-Packing"-Problem können über dieses Transportnetz maximal vier Kisten kollisionsfrei transportiert werden. Die folgenden vier generierten Pfade repräsentieren den Weg, den vier

Kisten über dieses Flussnetz nehmen. Die Auslastung der einzelnen Kanten können in der Abbildung anhand der Fluss- und Kapazitätswerte betrachtet werden.

$p_1 =< (q,a),(a,s) >$
$p_2 =< (q,a),(a,b),(b,c),(c,s) >$
$p_3 =< (q,c),(c,s) >$
$p_4 =< (q,c),(c,s) >$

Jede weitere Kiste, die in das Transportnetz gegeben wird, muss ebenfalls eine der drei Kanten überqueren, die den minimalen Schnitt bilden und würde die jeweilige Kapazität dieser Kante überlasten.

2.3 Operations Research

Das Path-Packing kann aus mathematischer Sicht als Optimierungsproblem aufgefasst werden, bei dem es das Ziel ist, den Gesamtnutzwert durch Einhaltung der Kantenkapazitäten zu maximieren. Seit vielen Jahrzehnten existieren solche Fragestellungen, die im Allgemeinen nach einem Optimum für ein gesetztes Ziel suchen. Die dabei verwendeten Methoden wurden anfänglich während des zweiten Weltkrieges für militärische und strategische Zwecke angewandt. Später ist erkannt worden, dass solche Fragestellungen auch im ökonomischen, betriebswirtschaftlichen und volkswirtschaftlichen Bereich zu finden sind. So entwickelte sich die Wissenschaft des „Operations Research". Heutzutage werden unter diesem Begriff nicht nur Optimierungsprobleme an sich zusammengefasst, sondern auch die mittlerweile enorm gewachsene Menge an mathematischen Methoden und den dazugehörigen Fachbegriffen (vgl. Koop u. Moock 2008, S. 1ff.; Zimmermann u. Stache 2001, S. 1ff.).

Graphentheorie, Monte-Carlo-Simulation, Spieltheorie, aber auch die Lineare Optimierung sind mathematische Methoden, die laut (Koop u. Moock 2008, S. 2) zu dieser Menge gehören. Die zuletzt genannte Methode der Linearen Optimierung dient als Grundlage für die algorithmischen Ansätze des Path-Packings und wird im Folgenden vorgestellt.

2.4 Lineare Optimierung

Die Vorgehensweise der Linearen Optimierung wird in diesem Kapitel näher betrachtet. Die beiden wichtigen Aspekte sind dabei die Vorstellung und Notation eines Linearen Programms und die Betrachtung des komplexitätstheoretischen Hintergrundes. Weiterhin findet das GlpK-Toolkit Erwähnung, welches zur praktischen Umsetzung des Path-Packings verwendet wird.

2.4.1 Lineares Programm

Ein lineares Programm (LP) dient zur Modellierung von Optimierungsproblemen und besteht einerseits aus einer Zielfunktion und andererseits aus dazugehörigen Nebenbedingungen. Die Notation eines LPs wird, basierend auf (Brucker 2007, S. 11f.), wie folgt festgehalten:

$$minimize \quad \sum_{i=1}^{n} c_i \, x_i$$

$$a_1 x_1 + \cdots + a_{1n} x_n \geq b_1$$
$$a_2 x_1 + \cdots + a_{2n} x_n \geq b_2$$
$$\vdots$$
$$a_m x_1 + \cdots + a_{mn} x_n \geq b_m$$

$$x_i \geq 0 \quad i = 1, \ldots, n$$

Dieses Beispiel stellt eine Zielfunktion dar, bei der das Minimum gesucht ist. Alternativ kann eine Zielfunktion auch als maximierend definiert sein, welche allerdings durch eine Multiplikation mit dem Faktor -1 in die dargestellte Minimierungsform transformiert werden kann (Syslo u. a. 2007, S. 3). Eine Zielfunktion liefert das gesuchte Optimum unter der Einhaltung von bestimmten Nebenbedingungen. Alle Vektoren $(x_1, x_2, \ldots, x_n)$, die dieses Gleichungssystem erfüllen, sind zulässige Lösungen. Unter diesen Vektoren ist einer zu finden, der das tatsächliche Minimum (oder Maximum) der Zielfunktion liefert und somit die Optimallösung.

2.4.2 Varianten der Linearen Optimierung

Ein wichtiger Faktor für das Lösen der Linearen Optimierung, vor allem hinsichtlich der praktischen Einsatzfähigkeit, ist die Laufzeit eines verwendeten Algorithmus. Im Laufe der Jahre wurden etliche mathematische Ansätze publiziert, die auf unterschiedlichen Eigenschaften von LPs basieren. Allgemein können folgende Varianten der Optimierungsmodellierung durch LPs unterschieden werden:

- *Lineare Optimierung*. Hierbei handelt es sich um die allgemeine Form der Linearen Programmierung. Die Variablen x_i eines LP besitzen keine ganzzahlige Beschränkung. Für solche Fälle wurde gezeigt, dass das Lösen dieser LPs in einem polynomiellen Aufwand möglich ist. Dies ist erstmalig in (Khachiyan 1980) auf Basis der Ellipsoidmethode gezeigt worden. Dieser Algorithmus ist zwar der theoretische Beweis für den polynomiellen Aufwand, allerdings wird der Simplex-Algorithmus, trotz seines exponentiellen Aufwandes im Worst Case, in der Praxis bevorzugt verwendet. Anwendung findet er deshalb, weil der Nachteil des exponentiellen Aufwandes nur selten auftaucht (vgl. Koop u. Moock 2008, S. 99ff.).

- *Ganzzahlige Optimierung*. Die in einem solchen LP verwendeten Variablen x_i besitzen eine ganzzahlige Beschränktheit. In vielen praktischen Anwendungen wird dies verlangt; bspw. bei der Produktion einer bestimmen Anzahl von Gegenständen oder bei der Berechnung einer benötigten Anzahl an Arbeitskräften.
  Diese mathematische Einschränkung des Zahlenraums für die Variablen x_i hat zur Folge, dass ein exponentieller Aufwand für das Lösen des LPs benötigt werden kann. Ein LP, bei dem die Variablen x_i nur zum Teil ganzzahlig sein müssen, wird als „mixed integer programming" (MIP) bezeichnet. Optimal arbeitende Verfahren sind bspw. das Branch & Bound Verfahren, das Schnittebenenverfahren und das Branch & Cut Verfahren (vgl. Nickel u. a. 2011, S. 183, 204ff.; Koop u. Moock 2008, S. 153).

- *Binäre Optimierung*. Die Variablen x_i eines LPs werden nur auf die numerischen Werte 0 und 1 beschränkt. Anwendungsfelder sind in einem solchen Fall bspw. die Modellierung von logischen Zusammenhängen. Die 0-1 Integer Programmierung, eines der klassischen NP-vollständigen Probleme, basiert ebenfalls auf der binären Optimierung (vgl. Karp 1972, S. 94; Nickel u. a. 2011, S. 191; Syslo u. a. 2007, S. 100f.).
  Auch die Modellierung des Path-Packings fällt in diese Kategorie. Ein Pfad wird entweder genommen ($x_i = 1$) oder er wird nicht genommen ($x_i = 0$). Für eine optimale Lösung kann ein Pfad aber nicht zu einem bestimmten „Anteil" genommen werden.

- *Relaxierte Optimierung*. Da ganzzahlige Optimierungsprobleme schwer zu lösen sind, kann Abhilfe verschafft werden, indem die Ganzzahligkeitsbedingungen der Variablen x_i gelöst werden. Beispielsweise werden die Bedingungen $x_i \in \mathbb{N}_0$ durch $x_i \geq 0$ und $x_i \in \{0, 1\}$ durch einen kontinuierlichen Zahlenbereich $0 \leq x_i \leq 1$ ersetzt. Das Ergebnis ist somit ein LP in allgemeiner Form und kann in polynomieller Zeit gelöst werden. Anhand der kontinuierlichen Lösungen wäre es möglich ganzzahlige Lösungen zu bestimmen, bspw. durch das Runden auf die nächste Ganzzahl. Hierbei muss allerdings darauf geachtet werden, dass die gerundeten Lösungen überhaupt in der zulässigen Lösungsmenge liegen. Weiterhin kann eine relaxierte Lösung auch weit von der eigentlichen ganzzahligen Lösung entfernt liegen. Daher ist diese Herangehensweise nur mit Vorsicht zu verwenden (vgl. Nickel u. a. 2011, S. 185; Koop u. Moock 2008, S. 153f.).

2.4.3 Verfahren zum Lösen von LPs

Wie im vorhergehenden Abschnitt erwähnt, existieren unterschiedliche algorithmische Ansätze für das Lösen von LPs. Die für diese Arbeit wichtigsten Algorithmen sollen kurz beschrieben werden, ohne den Schwerpunkt auf mathematische Details zu legen.

- *Simplex-Algorithmus* : Die zulässige Lösungmenge eines LPs kann als Schnittmenge von Halbräumen beschrieben werden; diese wird auch als konvexes Polyeder bezeichnet. Extrempunkte aus der Lösungsmenge M werden auch als Ecken des Polyeders bezeichnet. Es kann daher gefolgert werden: Wenn ein LP einen optimalen Punkt besitzt, so hat auch mindestens eine Ecke des Polyeders einen optimalen Punkt. Der Simplex-Algorithmus nutzt diese Ecken eines Polyeders aus, indem er zunächst eine beliebige Ecke wählt. Ausgehend von dieser versucht er benachbarte Ecken zu finden, die einen kleineren Funktionswert besitzen unter der Voraussetzung, dass es sich um eine Minimierungsfunktion handelt. Dieser Schritt wird solange wiederholt, bis keine absteigenden Kanten mehr gefunden werden können. Der Optimalpunkt ist somit gefunden. (Jarre u. Stoer 2004, S. 23; Nickel u. a. 2011, S. 14ff.)

- *Branch & Bound - Verfahren* : Dieses Verfahren besteht aus zwei wesentlichen Schritten: dem Branching und dem Bounding. Das Ziel des Branching ist es, ein Problem in Teilprobleme aufzuteilen. Dabei wird auch die Lösungsmenge des ursprünglichen Problems disjunkt in Teillösungsmengen unterteilt. Durch die Verzweigung eines Problems kann ein Lösungsbaum mit den Teilproblemen aufgebaut werden. Die Erwartung ist, dass Teilprobleme leichter zu lösen sind. Ein Problem kann nach unterschiedlichen Kriterien unterteilt werden. Die anschließende Frage ist, wie ein Teilproblem nun aus dem Verzweigungsbaum ausgewählt wird; dies wird auch als Backtracking bezeichnet. Ein Nebeneffekt ist, dass Teilprobleme durch das Bounding für weitere Unterteilungen ausgeschlossen werden können, wenn die Optimallösung nicht in diesem Teilproblem liegt. Dies wird mit dem Vergleich von Schranken umgesetzt. Eine globale untere Schranke F^u ist für jedes Teilproblem gültig. Jedes Teilproblem wiederum besitzt eine lokale obere Schranke F_i^l. Wenn $F_i^u \leq F^l$ gilt, so kann eine Optimallösung in diesem Teilproblem ausgeschlossen werden. Die globale Schranke wird aktualisiert, wenn ein zulässiger Punkt gefunden wird, der zu einem größeren Funktionswert führt (vgl. Nickel u. a. 2011, S. 205ff.).

- *Schnittebenen - Verfahren* : Eine alternative Möglichkeit ganzzahlige LPs zu lösen sind Schnittebenen-Verfahren. Bei diesem Ansatz wird ein ganzzahliges LP zunächst in der relaxierten Version gelöst. Die vorhandenen Nebenbedingungen werden durch das Hinzufügen weiterer Gleichungen – auch Schnittebenen genannt – erweitert; die Lösungsmenge wird folglich eingeschränkt. Dieser Schritt wird wiederholt, bis eine ganzzahlige Lösung gefunden wird. Die hinzugefügten Gleichungen werden durch die ganzzahligen Lösungen weiterhin erfüllt. Die relaxierte Lösung erfüllt diese allerdings nicht mehr, so dass diese für die Berechnung nicht mehr mit einbezogen werden (vgl. Nickel u. a. 2011, S. 213f.). Auch für Schnittebenenverfahren existieren unterschiedliche Ansätze, die bspw. von der GlpK-Bibliothek angeboten werden.

- *Branch & Cut Verfahren* : Das Branch & Cut-Verfahren ist eine Kombination aus dem Branch & Bound-Verfahren und dem Schnittebenen-Verfahren. In jedem Teilproblem des Lösungsbaumes werden Schnittebenen hinzugefügt, um so die Lösungsmenge des jeweiligen Teilproblems einschränken zu können (vgl. Nickel u. a. 2011, S. 220f.).

2.4.4 Gnu Linear Programming Toolkit

Das „GNU Linear Programming Toolkit" (GlpK) bietet die Möglichkeit, lineare Programme auf Basis der erwähnten Algorithmen zu lösen (Makhorin 2012). Neben dem direkten Einbinden der auf der Sprache C basierenden Bibliothek enthält diese auch einen Stand-Alone-Solver, in dem lineare Programme in Form des Cplex-Formates eingelesen werden können. Konkrete Verfahren und die dazugehörigen GlpK-Parameter für das Branch & Bound-Verfahren und den Schnittebenenverfahren können in Kapitel 4.3.3 im Kontext der Path-Packing-Analyse nachvollzogen werden.

2.5 Optimierungsprobleme in der Logistik

Dieser Abschnitt beschäftigt sich mit der theoretischen Sicht des Path-Packing-Problems und legt Gemeinsamkeiten mit anderen Optimierungsproblemen dar. Zu jedem Optimierungsproblem existiert ein zugehöriges Entscheidungsproblem. Wenn ein Entscheidungsproblem in der Klasse NPC liegt, so ist das dazugehörige Optimierungsproblem NP-schwer (vgl. Nickel u. a. 2011, S. 201f.). Die in diesem Abschnitt erwähnten Optimierungsprobleme sind aus komplexitätstheoretischer Sicht meist NP-schwer und folglich nur für kleine Problemgrößen in einer akzeptablen Zeit optimal zu lösen.

2.5.1 Rucksack-Problem

Das Rucksack-Problem (engl. knapsack-problem) betrachtet die Fragestellung, welche Gegenstände in einen Rucksack gepackt werden können. Dabei müssen zwei Eigenschaften gelten: Zum einen darf das Volumen des Rucksacks nicht überlastet werden und zum anderen muss der summierte Nutzwert der genommenen Gegenstände maximal werden. Mathematisch gesehen kann dieses Problem als LP notiert werden (Martello u. Toth 1990, S. 1f.).

Der Rucksack besitzt das Volumen c. Es existieren n Gegenstände, die jeweils von einer bool'schen Variable x_i mit $1 \leq i \leq n$ repräsentiert werden. Weiterhin wird jedem Gegenstand eine Variable p_i, die den Nutzwert darstellt, zugeordnet sowie die Variable w_i für das jeweilige Volumen des Gegenstandes. Die Variable x_i besitzt den Wert 1 gdw. der Gegenstand in den Rucksack gelegt wird; alternativ den Wert 0, wenn dieser nicht genommen wird.

Die Zielfunktion des LP wird daher wie folgt definiert:

$$maximize \sum_{i=1}^{n} p_i \, x_i$$

Die zu erfüllende Nebenbedingung lautet:

$$\sum_{i=1}^{n} w_i \, x_i \leq c$$

Nach (Karp 1972, S. 100) gehört das Rucksack-Problem zu den klassischen NP-vollständigen Problemen. Durch einen analogen Vergleich zwischen dem Path-Packing und dem Rucksack-Problem, kann gezeigt werden, dass das Path-Packing-Problem ebenfalls in die Klasse der NP-vollständigen Probleme fallen muss und somit mindestens genau so schwer zu lösen ist wie das Rucksack-Problem.

Das Gegenstück zum Volumen eines Gegenstandes beim Rucksack-Problem ist die Belegung eines Pfades auf einer Kante. Wenn ein Pfad genommen wird, so steigt das aktuelle Volumen – bzw. der aktuelle Flusswert – dieser Kante an. Da die Bedingung $f(e) \leq c(e)$ innerhalb eines Flussnetzes gelten muss, entspricht die Kapazität c einer Kante dem Volumen eines Rucksacks. Realistisch betrachtet, ist die Kantenanzahl E eines Flussnetzes größer 1, sodass das Path-Packing-Problem als eine Sammlung mehrerer Instanzen des Rucksack-Problems betrachtet werden kann. Jede Kante entspricht somit einem Rucksack und folglich auch einer eigenen Nebenbedingung, nämlich dass die Kapazität dieser Kante nicht überschritten werden darf. Im Vergleich dazu hat das Rucksack-Problem insgesamt nur eine Nebenbedingung.

Die Zielfunktionen können bei diesen beiden Problemen als identisch angesehen werden. Bei beiden Problemen werden die Summen der genommenen Nutzwerte maximiert: beim Rucksack-Problem hinsichtlich der genommenen Gegenstände; beim Path-Packing hinsichtlich der genommenen Pfade.

2.5.2 0-1 Integer-Programmierung

Karp (Karp 1972, S. 97) zeigt ebenfalls, dass das Problem der 0-1 Integer Programmierung in der Klasse NPC liegt. Bei diesem Problem wird ein binärer Vektor gesucht, der die Gleichung $Cx = d$ erfüllt. Dabei

sei C eine ganzzahlige Matrix und d ein ganzzahliger Vektor. Auch hierbei lässt sich eine überlappende Schnittmenge zum Path-Packing erkennen: Das Path-Packing-Problem kann als LP ausgedrückt werden (vgl. Kap. 3.3.1), bei dem die Variablen x_i genau einen solchen binären Vektor bilden. Die von Karp erwähnte Matrix C und der Vektor d werden durch die Nebenbedingungen geformt. Es wird der binäre Vektor gesucht, der den Gesamtnutzwert maximiert.

2.5.3 Bin-Packing-Problem

Ein weiteres NP-vollständiges Problem, welches Gemeinsamkeiten mit dem Path-Packing hat, ist das „Bin-Packing". Laut (Garey u. Johnson 1979, S. 226) existiert bei dieser Problembeschreibung eine endliche Menge U mit Gegenständen, die wiederum eine bestimmte Größe $s(u) \in \mathbb{Z}^+$ besitzen. Weiterhin existiert eine Zahl $B \in \mathbb{Z}^+$, die die Größe eines Behälters (engl. bin) beschreibt und eine Zahl $k \in \mathbb{Z}^+$, die der Gesamtanzahl an Behältern entspricht. Das Ziel ist es, die Gegenstände in die Behälter U_1 bis U_k so zu verteilen, dass die Summe der Größen der Gegenstände in einem Behälter nicht größer als die Behältergröße B ist. Bei der Verteilung der Gegenstände muss die Disjunktheit der Mengen $U_1, \ldots, U_k$ ebenfalls gelten. Eine Variante dieses Problem ist die Frage nach der minimalen Anzahl an Behältern (Wegener 2003, S. 17).

Die Definition des Path-Packing muss für einen Vergleich ein wenig erweitert werden. Zu diesem Zeitpunkt ist die Frage des Path-Packings so definiert, dass nur eine bestimmte Teilmenge aus der vorhandenen Pfadmenge P durch einen Algorithmus ausgewählt wird. Die Frage, was mit den übrigen Pfaden allerdings passiert, ist unbeantwortet geblieben. Das Path-Packing wird für diesen Gedankengang erweitert, indem nicht nur eine einmalige Teilmenge gewählt wird, sondern so viele Teilmengen, so dass alle Pfade einmal gewählt worden sind. Daher wird der Begriff des *Durchlaufes* eingeführt. Alle Kisten, die zu einem Durchlauf gehören, können wie bisher ohne Kollision auf das Transportnetz geschickt werden. Die Kisten werden gemäß den Pfaden eines Durchlaufes auf das Transportband gegeben, abgearbeitet und verlassen durch die Senke das Netz. Anschließend können die Kisten des nächsten Durchlaufes auf das Transportnetz gegeben werden.

Das Bin-Packing kann auf Basis dieser Erweiterung auf das Path-Packing übertragen werden. Das Pendant der Gegenstände im Bin-Packing-Problem sind die Pfade des Path-Packings, denn diese sollen nun auf verschiedene Durchläufe aufgeteilt werden. Wegener (Wegener 2003, S. 17) stellt die Frage nach der minimalen Anzahl an Kisten; beim Path-Packing wird dabei die minimale Anzahl an Durchläufen gesucht. Ein Durchlauf entspricht daher einem Behälter, in dem die Pfade abgelegt werden. Die Größe B eines Behälters kann mit den Kantenkapazitäten des Flussnetzes gleichgesetzt werden. Ein Behälter ist dann überfüllt, wenn die Summe der Gegenstandsgrößen die Behältergröße übersteigt. Übertragen auf das Path-Packing heißt dies, dass das Flussnetz genau dann überfüllt ist, wenn mindestens ein Flusswert einer Kante die jeweilige Kantenkapazität übersteigt.

3 Heuristiken und Algorithmen

Auf Basis der in Kapitel 2 beschriebenen Grundlagen soll sich dieses Kapitel mit den algorithmischen Ansätzen für das Path-Packing beschäftigen. Diese Ansätze werden zunächst allgemein definiert, in einem Beispiel veranschaulicht und anschließend einer Laufzeitanalyse unterzogen.

3.1 Greedy-Heuristik

Der erste Ansatz ist die Definition einer Greedy-Heuristik. Da NP-vollständige Optimierungsprobleme, zumindest für die Annahme $\mathsf{P} \neq \mathsf{NP}$, für das optimale Lösen eine exponentielle Laufzeit besitzen, ist eine Greedy-Heuristik eine Möglichkeit für das Berechnen einer Näherungslösung. Die Implementierung einer solchen Heuristik ist zwar effektiv und somit in Polynomialzeit zu realisieren, allerdings muss die dadurch berechnete Lösung nicht mit der Optimallösung übereinstimmen (vgl. Turau 2009, S. 47).

3.1.1 Definition und Ablauf

Die dargestellten Algorithmen 1 und 2 zeigen die definierte Greedy-Heuristik für das Path-Packing. Ein Flussnetz F und eine dadurch generierte Pfadmenge P sind die Eingabeparameter für diese Heuristik. Das Ergebnis ist eine Pfadmenge Q, in der alle Pfade gespeichert werden, die von der Heuristik gewählt worden sind. In Fachkreisen wird dieses Auswählen auch als „Pick"-Prozess bezeichnet.

```
1:  procedure GREEDY(Flownet F, Pathset P)
2:      Q ← NULL                                  ▷ leere Liste für genommene Pfade
3:      P ← Sort_Descending_By_Values(P)
4:      for each path in P do
5:          if ( Check_Path(F, path) == true ) then   ▷ Überprüfe, ob Pfad ins Flussnetz passt
6:              Q ← path
7:          end if
8:      end for
9:      TotalValue ← 0                            ▷ Summe der genommenen Nutzwerte
10:     for each q in Q do
11:         TotalValue = TotalValue + q.value
12:     end for
13: end procedure
```

Algorithmus 1: Greedy-Algorithmus

Zunächst wird die gegebene Pfadmenge P anhand der Nutzwerte der einzelnen Pfade absteigend sortiert (Alg. 1 Zeile 3). Für diese Operation kann ein gewöhnlicher Sortiertalgorithmus verwendet werden. Die Pfadmenge wird daraufhin iterativ durchlaufen und für jeden Pfad durch den Methodenaufruf $Check\_Path$ geprüft, ob dieser noch in das Flussnetz passt (Zeilen 4 bis 8). Letztendlich wird der Gesamtnutzwert berechnet, indem über die Pfadmenge der genommenen Pfade iteriert wird und die Nutzwerte aufsummiert werden (Zeilen 9 bis 12).

Die Methode $Check\_Path$ prüft, ob ein aktuell ausgewählter Pfad noch in das Flussnetz passt. Dazu erhält diese Methode zwei Eingabeparameter: das Flussnetz F und den potentiell zu nehmenden Pfad $path$.

Der Pfad $path$ wird dann akzeptiert, wenn keine Kante auf Basis der aktuellen Belegung durch diesen Pfad überlastet wird. Im ersten Teil dieser Methode (Alg. 2 Zeilen 4 bis 18) wird geprüft, ob der Pfad $path$ eine Kante des Flussnetzes überlastet. Dazu wird über die Kantenmenge des Pfades iteriert (Zeile 4) und für jede dieser Kanten die potentielle Erhöhung des Flusswertes berechnet. Diese wird in der Variable $add\_flowvalue$ gespeichert (Zeilen 6 bis 10). Für zyklische Pfade kann eine Kante des Pfades den Flusswert einer Flussnetzkante um mehr als den Wert 1 erhöhen. Daraufhin wird geprüft, ob die Summe aus dem aktuellen Flusswert einer Kante f und der Variable $add\_flowvalue$ die Kapazität einer Kante überlastet (Zeile 13). Ist dies der Fall, wird der Pfad verworfen. Wird ein Pfad akzeptiert, so wird der zweite Teil der Methode durchlaufen (Zeilen 19 bis 25). Dabei werden die Flusswerte der Kanten, die den Pfad bilden, tatsächlich erhöht.

```
1:  procedure CHECK_PATH(Flownet F, Path path)
2:      pathedges = p.edgeSet
3:      flownetedges = F.edgeSet
4:      for each e in pathedges do
5:          add_flowvalue ← 0                          ▷ potentieller Flusswert durch den neuen Pfad
6:          for each f in pathedges do
7:              if (e == f) then
8:                  add_flowvalue++
9:              end if
10:         end for
11:         for each f in flownetedges do              ▷ Prüfe, ob Kante überlastet wird
12:             if (e == f) then
13:                 if (f.flowvalue + add_flowvalue > f.capacity) then
14:                     return false                   ▷ Pfad wird nicht akzeptiert
15:                 end if
16:             end if
17:         end for
18:     end for
19:     for each e in pathedges do                     ▷ Erhöhung der jeweiligen Flusswerte
20:         for each f in flownetedges do
21:             if (e == f) then
22:                 f.flowvalue++
23:             end if
24:         end for
25:     end for
26:     return true                                    ▷ Pfad wird akzeptiert
27: end procedure
```

Algorithmus 2: Pfad-Überprüfen Methode des Greedy-Algorithmus

3.1.2 Anwendung der Greedy-Heuristik

Die Anwendung und Berechnung dieser Heuristik soll in einem Beispiel verdeutlicht werden. Als Grundlage dient das in Abbildung 3.1a dargestellte Flussnetz. Für diese Probleminstanz wird eine Pfadmenge $P = \{p_1, \ldots, p_{10}\}$ mit jeweils einem zugeordneten Nutzwert generiert. Die Pfadmenge wird durch die einmalig aufzurufende Sortieroperation nach ihrem Nutzwert sortiert. Die sortierten Pfade können in Tabelle 3.1 betrachtet werden:

Nutzwert	Pfad
92	$p_1 =< (q,a),(a,c),(c,s) >$
90	$p_2 =< (q,a),(a,b),(b,c),(c,s) >$
89	$p_3 =< (q,b),(b,c),(c,s) >$
66	$p_4 =< (q,b),(b,c),(c,s) >$
51	$p_5 =< (q,b),(b,c),(c,s) >$
50	$p_6 =< (q,a),(a,b),(b,c),(c,s) >$
41	$p_7 =< (q,b),(b,c),(c,s) >$
37	$p_8 =< (q,a),(a,s) >$
29	$p_9 =< (q,b),(b,c),(c,s) >$
19	$p_{10} =< (q,a),(a,c),(c,s) >$

Tab. 3.1: Sortierte Pfadmenge für Greedy-Heuristik

Diese sortierte Pfadmenge wird iterativ durchlaufen und für jeden Pfad geprüft, ob dieser noch in das Flussnetz passt. Die jeweils genommenen Pfade werden in der folgenden Abbildung 3.1 farblich gekennzeichnet. Das Ergebnis dieser Iterationen sieht wie folgt aus:

- Pfad p_1 wird akzeptiert, da keine Kanten anfänglich belastet sind (Abb. 3.1b).
- Pfad p_2 wird akzeptiert, da die Kanten (q,a), (a,b), (b,c) und (c,s) nicht voll belastet sind (Abb. 3.1c).
- Pfad p_3 wird akzeptiert, da die Kanten (q,b), (b,c) und (c,s) nicht voll belastet sind (Abb. 3.1d).
- Pfad p_4 wird nicht akzeptiert, da die Kante (b,c) ihre Höchstkapazität erreicht hat.
- Pfad p_5 wird nicht akzeptiert, da die Kante (b,c) ihre Höchstkapazität erreicht hat.
- Pfad p_6 wird nicht akzeptiert, da die Kante (b,c) ihre Höchstkapazität erreicht hat.
- Pfad p_7 wird nicht akzeptiert, da die Kante (b,c) ihre Höchstkapazität erreicht hat.
- Pfad p_8 wird akzeptiert, da die Kanten (q,a) und (a,s) nicht voll belastet sind (Abb. 3.1e).
- Pfad p_9 wird nicht akzeptiert, da die Kante (b,c) ihre Höchstkapazität erreicht hat.
- Pfad p_{10} wird akzeptiert, da die Kanten (q,a), (a,c) und (c,s) nicht voll belastet sind (Abb. 3.1f).

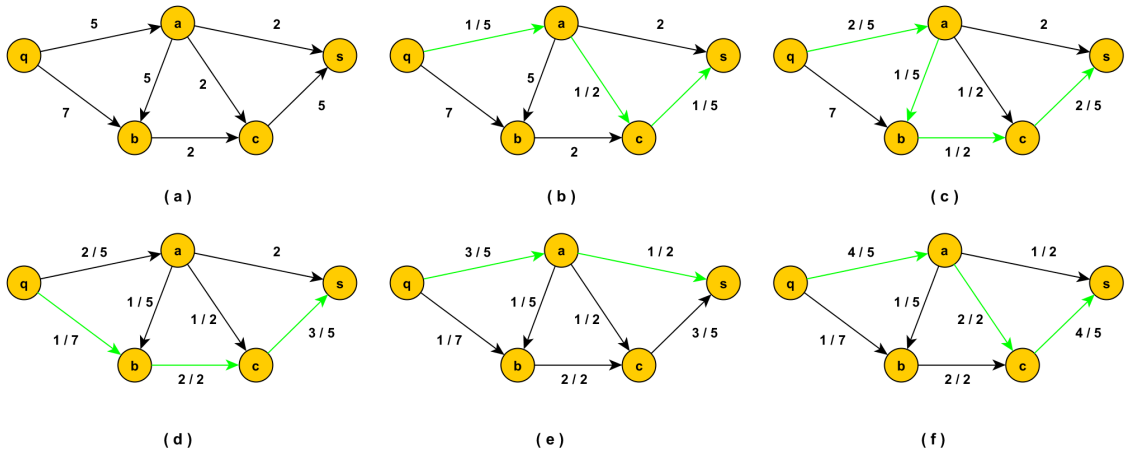

Abb. 3.1: Anwendung der Greedy-Heuristik

Abbildung 3.1f zeigt die Endbelegung der Kanten. Der minimale Schnitt des Flussnetzes ist gegeben durch die Kanten (a,s), (a,c) und (b,c) und besitzt den Wert $2+2+2 = 6$. Obwohl der minimale Schnitt eine obere

Grenze an gleichzeitig zu transportierenden Kisten darstellt, ist zu erkennen, dass, dieser nicht vollständig ausgenutzt wird. Die Summe der Flusswerte dieser Kanten beträgt $1 + 2 + 2 = 5$.

Des Weiteren kann der Gesamtnutzwert, der durch die Greedy-Heuristik erreicht wird, berechnet werden. Für diese Probleminstanz und die fünf somit ausgewählen Pfaden ergibt sich der Gesamtnutzwert von $92 + 90 + 89 + 37 + 19 = 327$.

3.1.3 Laufzeitanalyse

Dieser Abschnitt beschäftigt sich mit der Bestimmung der Laufzeit der vorgestellten Heuristik, die im Worst Case auftritt. Wichtig ist nicht nur zu wissen, dass eine Heuristik in polynomieller Laufzeit ausgeführt werden kann, sondern auch wie hoch der Grad dieses Polynoms ist. Obwohl eine polynomielle Laufzeit vorliegt, kann der Grad des Polynoms allerdings so hoch sein, dass die Heuristik mit einem solchen Polynom dennoch in der Praxis als nicht mehr rentabel gilt.

Für die Bestimmung der Laufzeit wird daher festgelegt, dass alle Zuweisungen, Inkrementierungen von Variablen und Initialisierungen von Datenstrukturen, wie bspw. Listen, einen Aufwand von $\mathcal{O}(1)$ besitzen (vgl. Cormen u. a. 2001, S. 17 und 45). Auf Basis des gegebenen Flussnetzes $F(V, E)$ und der gegebenen Pfadmenge P werden folgende Variablen festgelegt: Die Anzahl der Knoten des Flussnetzes $n = |V|$, die Kanten des Flussnetzes $m = |E|$, die Anzahl der Pfade $p = |P|$ und die Länge des längsten Pfad $pathlength_{max}$, die durch die Anzahl der Kanten in einem Pfad bestimmt wird.

Die Gesamtlaufzeit der Greedy-Heuristik wird bestimmt, indem folgende drei Methoden untersucht werden:

- Sortieroperation : Die Heuristik verwendet eine Sortieroperation, die durch einen effizienten Sortieralgorithmus, wie bspw. den Quicksort-Algorithmus, umgesetzt werden kann. Dabei werden die Pfade p_i nach ihren Nutzwerten sortiert. Es kann gezeigt werden, dass ein effizienter Sortieralgorithmus eine Laufzeit von $\mathcal{O}(p \log p)$ besitzt (vgl. Ottmann u. Widmayer 2012, S. 81).

- $Check\_Path$-Methode : Diese Methode wird in Algorithmus 2 vorgestellt. Die Zeilen 6 bis 17 beschreiben zwei hintereinander ausgeführte Iterationen, die $pathlength_{max}$- bzw. m-mal durchlaufen werden. Diese beiden Iterationen werden wiederum $pathlength_{max}$-mal aufgerufen (Alg. 2 Zeile 4). Die Laufzeit kann daher mit

$$\mathcal{O}(pathlength_{max}) \left(\mathcal{O}(pathlength_{max}) + \mathcal{O}(m)\right)$$

$$= \mathcal{O}(pathlength^2_{max}) + \mathcal{O}(pathlength_{max})\, \mathcal{O}(m)$$

  beschrieben werden. Der zweite Teil der Methode (Zeilen 19 bis 25) besteht aus zwei ineinander geschachtelten Schleifen, die $pathlength_{max}$- bzw. m-mal durchlaufen werden. Die Laufzeit beträgt in diesem Teil $\mathcal{O}(pathlength_{max})\, \mathcal{O}(m)$. Die Summe beider Teile ergibt somit die Gesamtlaufzeit der $Check\_Path$-Methode:

$$\mathcal{O}(pathlength^2_{max}) + \mathcal{O}(pathlength_{max})\, \mathcal{O}(m) + \mathcal{O}(pathlength_{max})\, \mathcal{O}(m).$$

  In einer allgemeinen Probleminstanz können die erzeugten Pfade Zyklen enthalten. Daher können Pfade existieren, deren Anzahl an Kanten größer ist, als die Anzahl an Kanten im Flussnetz. Die Variable $pathlength_{max}$ ist somit eine obere Grenze für m. Auf Basis dieser Abschätzung kann die Laufzeit zusammengefasst werden zu

$$\begin{aligned} &\mathcal{O}(pathlength^2_{max}) + \mathcal{O}(pathlength^2_{max}) + \mathcal{O}(pathlength^2_{max}) \\ &= 3 \cdot \mathcal{O}(pathlength^2_{max}) \\ &\in \mathcal{O}(pathlength^2_{max}) \end{aligned}$$

  Die Ordnung der Laufzeit lautet folglich $\mathcal{O}(pathlength^2_{max})$.

- Greedy-Algorithmus : Wie in Algorithmus 1 ersichtlich, wird in diesem zunächst der Sortieralgorithmus aufgerufen. Die folgende Schleife (Alg. 1 Zeilen 4 bis 8) ruft p-mal die $Check\_Path$-Methode auf. Die Laufzeit der Schleife beträgt daher: $\mathcal{O}(p)\,\mathcal{O}(pathlength^2_{max})$. Dies ist gleichzeitig auch die Gesamtlaufzeit der vollständigen Greedy-Heuristik, da sowohl die Laufzeit des Sortieralgorithmus, als auch die Laufzeit der Berechnung des Gesamtnutzwertes (Zeilen 10 bis 12) von der Ordnung her nicht größer sind.

Die Gesamtlaufzeit ist somit zum einen quadratisch abhängig von der Länge des längsten Pfades und zum anderen von der Anzahl der Pfade. Es folgt somit die polynomielle Laufzeit des Gesamtalgorithmus.

Bei dieser Analyse wird allerdings vorausgesetzt, dass die gegebenen Pfade nicht zyklenfrei sind. Die Anzahl an Kanten in einem Pfad kann höher sein als die Anzahl an Kanten im Flussnetz selbst. Würde die Voraussetzung gelten, dass die gegebenen Pfade zyklenfrei sind, so kann die Länge eines Pfades $pathlength_{max}$ höchstens die Größe m erreichen. Für diesen Fall kann die Laufzeit beschrieben werden durch: $\mathcal{O}(p\,m^2)$.

3.2 Greedy-Heuristik mit spezifischem Nutzwert

Wird die randomisierte Generierung eines Flussnetzes für die Greedy-Heuristik genauer betrachtet, so fällt auf, dass auch Flussnetze erstellt werden können, die Zyklen enthalten. Auf Basis eines solchen zyklischen Graphen werden Pfade erstellt, die ebenfalls Zyklen enthalten können. Durch das Existieren von Zyklen ist es möglich, dass Kanten in Pfaden mehrfach vorkommen und folglich die Kantenkapazitäten für die Wahl weiterer Pfade ungünstig belasten. Durch das Vorgehen der Greedy-Heuristik können zyklische Pfade mit einem hohen Nutzwert akzeptiert werden, die allerdings die Kantenkapazitäten für andere Pfade – d.h. solche mit einem niedrigeren Nutzwert – blockieren. Global betrachtet wäre die Wahl von Pfaden mit niedrigeren Nutzwerten in der Summe dann aber eventuell besser, um einen größeren Gesamtnutzwert zu erreichen. Dieser beschriebene Nachteil einer Greedy-Heuristik soll ausgeglichen werden. Zyklische Pfade mit mehrfach belegten Kanten müssen als weniger nützlich für den Auswahlprozess der Greedy-Heuristik gelten. Dies wird durch die Einführung eines spezifischen Nutzwertes für jeden Pfad p erreicht.

3.2.1 Definition und Ablauf

Der spezifische Nutzwert ist folglich definiert als:

$p.spec\_value = \frac{p.value}{max(\{\#e_1,\#e_2,...,\#e_n\})}$ mit $\#e_i$ = Häufigkeit der Kante e_i im Pfad p

Der Ablauf dieser abgewandelten Greedy-Heuristik unterscheidet sich zur ursprünglichen Version nur in der Sortieroperation. Die Pfadmenge wird nicht mehr anhand ihrer Nutzwerte sortiert, sondern auf Basis der berechneten spezifischen Nutzwerte. Der Pseudocode der Greedy-Heuristik aus Kapitel 3.1.1 wird mit Ausnahme dieser anzupassenden Sortieroperation übernommen.

3.2.2 Anwendung der Greedy-Heuristik mit spezifischen Nutzwert

Auch dieser heuristische Ansatz soll durch ein geeignetes Beispiel veranschaulicht werden. Abbildung 3.2 zeigt das Flussnetz, welches in diesem Beispiel verwendet wird.

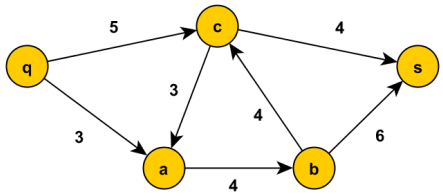

Abb. 3.2: Zyklisches Flussnetz für Greedy-Heuristik mit speziellem Nutzwert

Die Kanten (c,a), (a,b) und (b,c) bilden einen Zyklus. Die generierte Pfadmenge wird allerdings nach den spezifischen Nutzwerten sortiert, wie in Tabelle 3.2 ersichtlich:

Spezifischer Nutzwert	**Nutzwert**	**Pfad**
92	92	$p_1 =< (q,c),(c,s) >$
70	70	$p_4 =< (q,c),(c,s) >$
66	66	$p_5 =< (q,c),(c,a),(a,b),(b,s) >$
61	61	$p_6 =< (q,a),(a,b),(b,c),(c,s) >$
52	52	$p_7 =< (q,a),(a,b),(b,c),(c,s) >$
49	49	$p_8 =< (q,c),(c,a),(a,b),(b,s) >$
43	86	$p_2 =< (q,a),(a,b),(b,c),(c,a),(a,b),(b,s) >$
39	78	$p_3 =< (q,c),(c,a),(a,b),(b,c),(c,a),(a,b),(b,s) >$
29	29	$p_9 =< (q,c),(c,a),(a,b),(b,c),(c,s) >$
11	11	$p_{10} =< (q,a),(a,b),(b,c),(c,s) >$

Tab. 3.2: Sortierte Pfadmenge für Greedy-Heuristik mit spez. Nutzwert

Die Pfade p_2 und p_3 enthalten je einen Zyklus:

- Im Pfad p_2 ist die Kante (a,b) doppelt enthalten. Daher gilt: $\#e_{a,b} = 2$. Es ergibt sich der spezifische Nutzwert : $\frac{86}{2} = 43$
- Im Pfad p_3 sind die Kanten (a,b) und (c,a) doppelt enthalten. Daher gilt: $\#e_{a,b} = \#e_{c,a} = 2$. Es ergibt sich der spezifische Nutzwert : $\frac{78}{2} = 39$

Durch den jeweiligen spezifischen Nutzwert werden diese beiden Pfade in der Pfadliste eher in der unteren Hälfte einsortiert. Bei allen azyklischen Pfaden ist der spezifische Nutzwert gleich dem eigentlichen Nutzwert. Die nach dem spezifischen Nutzwert sortierte Pfadmenge wird iterativ durchlaufen und für jeden Pfad geprüft, ob dieser noch in das Flussnetz passt. Hierbei werden die genommenen Pfade erneut grün eingefärbt. Das Ergebnis dieser Iteration sieht wie folgt aus:

- Pfad p_1 wird akzeptiert, da keine Kanten anfänglich belastet sind. (Abb. 3.3a).
- Pfad p_4 wird akz., da die Kanten (q,c) und (c,s) nicht voll belastet sind (Abb. 3.3b).
- Pfad p_5 wird akz., da die Kanten (q,c), (c,a), (a,b) und (b,s) nicht voll belastet sind (Abb. 3.3c).
- Pfad p_6 wird akz., da die Kanten (q,a), (a,b), (b,c) und (c,s) nicht voll belastet sind (Abb. 3.3d).
- Pfad p_7 wird akz., da die Kanten (q,a), (a,b), (b,c) und (c,s) nicht voll belastet sind (Abb. 3.3e).
- Pfad p_8 wird akz., da die Kanten (q,c), (c,a), (a,b) und (b,s) nicht voll belastet sind (Abb. 3.3f).
- Pfad p_2 wird nicht akzeptiert, da die Kante (a,b) ihre Höchstkapazität erreicht hat.
- Pfad p_3 wird nicht akzeptiert, da die Kante (a,b) ihre Höchstkapazität erreicht hat.
- Pfad p_9 wird nicht akzeptiert, da die Kanten (a,b) und (c,s) ihre Höchstkapazitäten erreicht haben.
- Pfad p_{10} wird nicht akzeptiert, da die Kante (a,b) ihre Höchstkapazität erreicht hat.

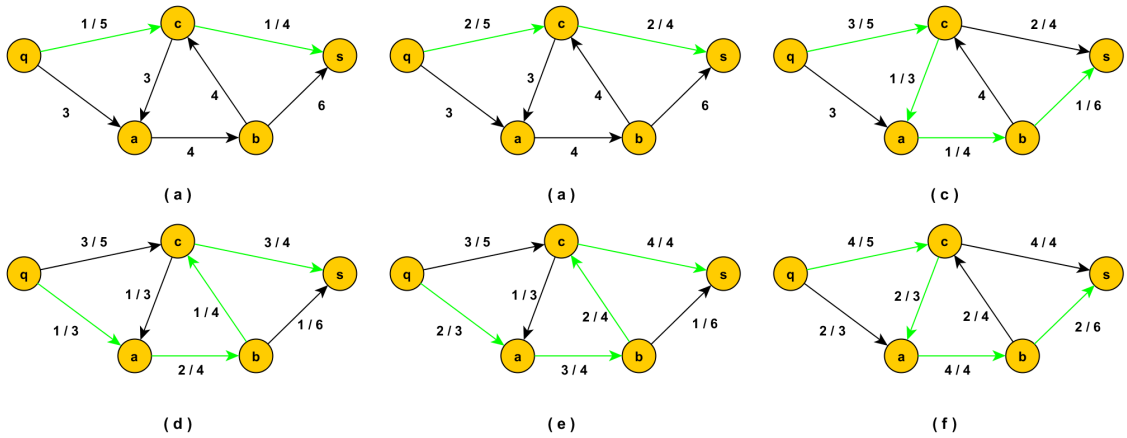

Abb. 3.3: Anwendung der Greedy-Heuristik mit spez. Nutzwert

Der Gesamtnutzwert ergibt sich durch die Summe der Nutzwerte der ausgewählten Pfade und beträgt: $92 + 70 + 66 + 61 + 52 + 49 = 390$. Für einen einfachen Vergleich kann ebenfalls die ursprüngliche Greedy-Heuristik auf diese Probleminstanz angewandt werden. So ergibt sich ein Gesamtnutzwert von $92+86+78+70 = 326$. Die zyklischen Pfade p_2 und p_3 würden laut dem Vorgehen dieser ursprünglichen Heuristik ausgewählt werden und sind der Grund dafür, dass die Pfade p_5 bis p_8 nicht mehr beansprucht werden können. Folglich kann für dieses Beispiel gezeigt werden, dass für die identische Probleminstanz die Greedy-Heuristik basierend auf dem spezifischen Nutzwert im Vergleich zur ursprünglichen Heuristik einen höheren Gesamtnutzwert erbringt.

3.2.3 Laufzeitanalyse

Der einzige Unterschied zwischen den beiden vorgestellten Heuristiken ist die Sortieroperation, die die Pfade nun nach ihrem spezifischen Nutzwert sortiert. Daher folgt, dass bei dieser modifizierten Heuristik dieselbe Laufzeit vorliegt wie bei der ursprünglichen Greedy-Heuristik.

3.3 Lineare Optimierung

Interessant sind aus wissenschaftlicher Sicht nicht nur Näherungslösungen, wie sie durch Heuristiken erreicht werden können, sondern auch die optimale Lösung einer Probleminstanz. Eine Möglichkeit, diese optimale Lösung zu berechnen, ist die Lineare Programmierung. Für das Path-Packing wird in diesem Kapitel ein binäres lineares Programm definiert und an einem Beispiel veranschaulicht.

3.3.1 Definition des linearen Programms

Das Path-Packing-Problem kann durch ein binäres, lineares Programm beschrieben werden (vgl. Kap. 2.4.2). Ein Pfad wird hierbei durch eine binäre Variable $x_i \in \{0, 1\}$ und einem Gewichtswert (bzw. Nutzwert) w_i repräsentiert. Die Variable x_i erhält den Wert 1 gdw. der Pfad durch die durchgeführte Berechnung genommen wird und 0 wenn dieser nicht genommen wird.

Für die Fragestellung des Path-Packings wird eine Teilmenge der Pfadmenge P gesucht, so dass der Gesamtnutzwert maximiert wird. Die Zielfunktion dieses linearen Programms lautet daher:

$$\max \sum_{i=1}^{p} w_i x_i \qquad x_i \in \{0, 1\} \qquad p = |P|$$

Für das Erreichen des maximalen Wertes der Zielfunktion muss die Nebenbedingung eingehalten werden, dass die Gesamtkapazität einer jeden Kante $c(e)$ nicht überschritten werden darf. Dies wird durch folgende

Nebenbedingung ausgedrückt:

$$\sum_{i=1}^{p} \chi_e(p_i) x_i \leq c(e) \qquad \forall e \in E$$

$$\chi_e(p_i) = \begin{cases} 1 & \text{falls Kante e in Weg } p_i \text{ vorkommt.} \\ 0 & \text{sonst} \end{cases}$$

Der Faktor $\chi_e(p_i)$ besitzt den Wert 1 gdw. eine Kante e im Pfad p_i vorkommt; andernfalls den Wert 0. Wenn eine Kante e in einem Pfad p_i vorkommt ($\chi_e(p_i) = 1$) und dieser Pfad p_i tatsächlich genommen wird ($x_i = 1$), so ergibt das Produkt dieser beiden Faktoren ebenfalls den Wert 1. Die Summe der Produkte $\chi_e(p_i)x_i$ bildet die aktuelle Belegung einer Kante e.

3.3.2 Anwendung des lineares Programms

Die Umsetzung und Berechnung des linearen Programms durch die GlpK-Bibliothek soll anhand eines Beispiels demonstriert werden. In diesem Beispiel dienen folgende Pfade (Tab. 3.3), die nicht auf Basis ihres Nutzwertes sortiert sein müssen, als Grundlage:

Nutzwert	Pfad
71	$p_1 =< (q,b),(b,c),(c,a),(a,s) >$
69	$p_2 =< (q,b),(b,c),(c,a),(a,s) >$
48	$p_3 =< (q,d),(d,c),(c,a),(a,s) >$
42	$p_4 =< (q,a),(a,b),(b,s) >$
83	$p_5 =< (q,b),(b,c),(c,a),(a,b),(b,c),(c,a),(a,s) >$
38	$p_6 =< (q,b),(b,c),(c,a),(a,s) >$
55	$p_7 =< (q,a),(a,b),(b,c),(c,a),(a,b),(b,s) >$
17	$p_8 =< (q,a),(a,b),(b,s) >$
07	$p_9 =< (q,d),(d,c),(c,a),(a,b),(b,c),(c,a),(a,s) >$
03	$p_{10} =< (q,a),(a,s) >$

Tab. 3.3: Verwendete Pfadmenge für das Beispiel der Linearen Programmierung

Die Berechnung durch den GlpK-Solver liefert das Ergebnis, dass die Pfade p_1 bis p_4, p_6 bis p_8 und p_{10} genommen werden. Das dabei verwendete Flussnetz kann in Abbildung 3.4 betrachtet werden. Des Weiteren sind in dieser Abbildung die Endbelegungen der Kanten durch die acht genommenen Pfade ersichtlich.

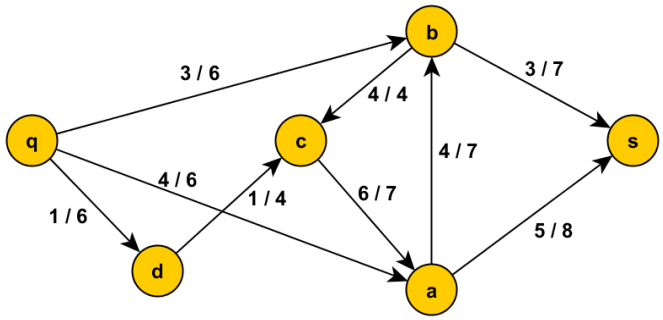

Abb. 3.4: Anwendung der linearen Programmierung

Durch das Aufsummieren der Nutzwerte der berechneten Pfade ergibt sich der Gesamtnutzwert von $71 + 69 + 48 + 42 + 38 + 55 + 17 = 343$ für den Ansatz der Linearen Programmierung. Dies ist die optimale Lösung für diese Probleminstanz. Im Vergleich zur Greedy-Heuristik und der modifizierten Greedy-Heuristik wird ersichtlich, dass der berechnete Wert des linearen Programms höher ist. Tabelle 3.4

Pfad	Nutzwert	LP	Greedy	spez. Greedy
p_1	71	x	x	x
p_2	69	x	x	x
p_3	48	x	x	x
p_4	42	x	x	x
p_5	83		x	x
p_6	38	x		
p_7	55	x		
p_8	17	x	x	x
p_9	07			
p_{10}	03	x	x	x
Gesamtnutzwert:		343	333	333

Tab. 3.4: Vergleich der ausgewählten Pfade der Verfahren

zeigt die Auswahl der einzelnen Pfade für diese Probleminstanz auf Basis der drei verschiedenen Ansätze sowie die Gesamtnutzwerte im Vergleich. In diesem Beispiel ist die berechnete Pfadmenge und folglich auch der Gesamtnutzwert der beiden Greedy-Heuristiken identisch.

3.3.3 Laufzeitanalyse

Laut der Dokumentation von GlpK (vgl. GnuProject 2014d, S. 61f.) werden MIPs durch das Branch & Cut-Verfahren gelöst. Wie in Kapitel 2.4.2 schon gezeigt, kann dieses Verfahren zur Lösung von ganzzahligen Problemen verwendet werden. Allerdings muss im Worst Case ein exponentieller Berechnungsaufwand erwartet werden (vgl. Nickel u. a. 2011, S. 204).

3.4 LP-basierte Heuristik

Sowohl der Ansatz der Greedy-Heuristik als auch der Ansatz der binären Optimierung haben jeweils Vor- und Nachteile. Die Heuristik besitzt zwar eine effizientere Laufzeit, kann aber nicht sicherstellen, dass eine Näherungslösung auch nah genug an der Optimallösung liegt. Bei der binären Optimierung ist dies genau umgekehrt: Nachteilhaft ist die exponentielle Laufzeit, wohingegen die berechnete Lösung die Optimallösung ist. Es wird nun versucht, die Vorteile beider Verfahren zu kombinieren.

Dieser Abschnitt beschäftigt sich mit dem beschriebenen Ansatz, ein LP ohne Ganzzahligkeitsbedingung in Polynomialzeit zu lösen und die kontinuierlichen Lösungen so zu interpretieren, dass eine qualitativ hochwertige Näherungslösung erreicht wird. Dieses Verfahren wird im Folgenden auch als LP-Heuristik bezeichnet.

3.4.1 Definition und Ablauf

Algorithmus 3 legt den Ablauf dieses heuristischen Ansatzes dar. Diese LP-Heuristik erwartet drei Parameter: Ein LPObject, welches durch die Anbindung der GlpK-Bibliothek vorliegt und die Bestandteile des LPs verwaltet, ein Flussnetz F sowie eine Pfadmenge P. Das Ergebnis ist wieder eine Liste Q, in der die genommenen Pfade gespeichert werden. Die Hilfsliste $OneZeroPaths$ dient dazu, die Pfade zwischenzuspeichern, die einen LP-Wert $0 < x_i < 1$ besitzen.

Die in Kapitel 3.3 beschriebene Zielfunktion und die dazugehörige Nebenbedingung des linearen Programms werden übernommen; allerdings wird die Ganzzahligkeitsbedingung $x_i \in \{0, 1\}$ in die Bedingung $0 \leq x_i \leq 1$ abgeändert. Durch das Lösen dieses LPs (Alg. 3 Zeile 4) wird jeder Variablen x_i ein Wert zwischen 0 und 1 zugewiesen.

```
1:  procedure RELAXLP(LPObject LP, Flownet F, Pathset P)
2:      Q ← NULL                                   ▷ leere Liste für genommene Pfade
3:      OneZeroPaths ← NULL                        ▷ leere Hilfsliste für Pfade mit LP-Wert zw. 0 und 1
4:      LP.findSolution()                          ▷ Löse relaxiertes LP
5:      for i = 1 to p do
6:          x_i ← LP.getLPvalue(i)                 ▷ LP-Wert für x_i auslesen
7:          if ( x_i ≥ 1 − ε ) then
8:              Q ← P.get(i)
9:          end if
10:         if ( x_i ≥ ε ) then
11:             OneZeroPaths ← P.get(i)
12:         end if
13:     end for
14:     for each path in Q do                      ▷ Übertrage genommene Pfade in Flussnetz F
15:         for each e in path.edgeSet do
16:             for each f in flownet.edgeSet do
17:                 if ( e == f ) then
18:                     f.IncValue()
19:                 end if
20:             end for
21:         end for
22:     end for
23:     ZeroOnePaths ← Sort_Descending_By_LPValues(ZeroOnePaths)
24:     for each path in OneZeroPaths do           ▷ Prüfe, ob Pfad noch genommen werden kann
25:         if ( Check_Path(F, path) == true ) then
26:             Q ← path
27:         end if
28:     end for
29:     TotalValue ← 0                             ▷ Summe der genommenen Nutzwerte
30:     for each q in Q do
31:         TotalValue = TotalValue + q.value
32:     end for
33: end procedure
```

Algorithmus 3: Ablauf der LP-Heuristik

Anschließend wird für jede Variable x_i geprüft, ob dieser Wert näherungsweise dem Wert 1 entspricht. Ist dies der Fall, so wird dieser Pfad genommen und in Liste Q gespeichert. Liegt dieser Wert zwischen 0 und 1, so wird dieser in die Hilfsliste $OneZeroPaths$ gespeichert. Alle restlichen Pfade besitzen näherungsweise den Wert 0 und werden somit automatisch verworfen (Zeilen 5 bis 13).

Die aktuellen Flusswerte der einzelnen Kanten im Flussnetz F werden nun auf Basis der in Liste Q bislang genommenen Pfade berechnet (Zeilen 14 bis 22). Die zwischengespeicherten Pfade, die einen LP-Wert zwischen 0 und 1 besitzen, werden nach diesen Werten absteigend sortiert (Zeile 23) und daraufhin geprüft, ob diese noch in die aktuelle Belegung des Flussnetzes passen (Zeilen 24 bis 28). Der letzte Schritt (Zeilen 29 bis 32) ist das Aufsummieren der Nutzwerte der genommenen Pfade.

Wegen Ungenauigkeiten der Arithmetik auf Basis von Fließkommazahlen in der Berechnung der GlpK-Bibliothek ist es möglich, dass Werte „$0 \pm \varepsilon$" bzw. „$1 \pm \varepsilon$" berechnet werden, anstatt den exakten numerischen Werten „0" bzw. „1". Diese Ungenauigkeit wird durch die Verwendung der Variable ε, die hinreichend klein gewählt wird, bei der Wahl der Pfade berücksichtigt.

3.4.2 Anwendung der LP-Heuristik

Auch dieser Ansatz soll anhand eines Beispiels veranschaulicht werden. Die Pfade sowie das Flussnetz aus Kapitel 3.3.2 werden hierfür wiederverwendet. Auf Grundlage dieser Testdaten ergibt sich folgende Zielfunktion :

$$maximize \quad 71x_1 + 69x_2 + 48x_3 + 42x_4 + 83x_5 + 38x_6 + 55x_7 + 17x_8 + 7x_9 + 3x_{10}$$

Diese lineare Funktion wird unter Einhaltung folgender Nebenbedingungen maximiert:

$$\begin{aligned}
x_1 + x_2 + x_5 + x_6 &\leq 4 \\
x_3 + x_9 &\leq 6 \\
x_4 + x_7 + x_8 + x_{10} &\leq 6 \\
x_4 + x_5 + 2\,x_7 + x_8 + x_9 &\leq 7 \\
x_3 + x_9 &\leq 4 \\
x_1 + x_2 + 2\,x_5 + x_6 + x_7 + x_9 &\leq 6 \\
x_4 + x_7 + x_8 &\leq 7 \\
x_1 + x_2 + x_3 + 2\,x_5 + x_6 + x_7 + 2\,x_9 &\leq 7 \\
x_1 + x_2 + x_3 + x_5 + x_6 + x_9 + x_{10} &\leq 8
\end{aligned}$$

$$0 \leq x_i \leq 1 \quad \forall \quad i \in \{1 \dots n\}$$

Wie in Kapitel 3.3 schon gezeigt worden ist, ergibt sich der Lösungsvektor $x = (1, 1, 1, 1, 0, 1, 1, 1, 0, 1)$ für die optimale ganzzahlige Lösung. Im Vergleich dazu, wird in diesem Beispiel der kontinuierliche Lösungsvektor $x = (1.0, 1.0, 1.0, 1.0, 0.5, 0.0, 1.0, 1.0, 0.0, 1.0)$ berechnet.

Auf Grundlage der Lösungsvektoren können daher die beiden Optimalwerte berechnet werden. Es wird festgestellt, dass der Optimalwert der kontinuierlichen Lösung größer ist, als der der ganzzahligen Lösung.

- Für die ganzzahlige optimale Lösung ergibt sich ein Nutzwert von :
  $71 \cdot 1 + 69 \cdot 1 + 48 \cdot 1 + 42 \cdot 1 + 83 \cdot 0 + 38 \cdot 1 + 55 \cdot 1 + 17 \cdot 1 + 7 \cdot 0 + 3 \cdot 1 = 343$
- Für die kontinuierliche Lösung ergibt sich ein Nutzwert von :
  $71 \cdot 1 + 69 \cdot 1 + 48 \cdot 1 + 42 \cdot 1 + 83 \cdot 0.5 + 38 \cdot 0 + 55 \cdot 1 + 17 \cdot 1 + 7 \cdot 0 + 3 \cdot 1 = 346.5$

Der kontinuierliche Lösungsvektor muss für die Bestimmung der finalen Pfadmenge noch weiter verarbeitet werden. Bei dem Vergleich einer ganzzahligen und kontinuierlichen Lösung kann der Fall auftreten, dass Pfade, die nicht Teil der Optimallösung sind, dennoch einen LP-Wert besitzen, der deutlich größer als 0 ist. In diesem Beispiel trifft dies auf Pfad x_5 zu. Bei der Optimallösung lautet der zugehörige LP-Wert 0; bei der kontinuierlichen Lösung 0.5. Auch ist es möglich, dass Pfade, die eigentlich Teil der Optimallösung sind, innerhalb der Näherungslösung den Wert 0.0 besitzen und somit für die finale heuristische Pfadmenge direkt ausgeschlossen werden. Auch dieser Fall ist bei diesem Beispiel zu erkennen. Pfad x_6 wird in der Optimallösung genommen; bei der kontinuierlichen Lösung besitzt dieser den Wert 0.0 und wird somit direkt verworfen.

Um anschließend die finale Pfadmenge als Lösung für das heuristische lineare Programm zu erhalten, werden alle Pfade, die näherungsweise den Wert 1 besitzen, in diese finale Pfadmenge genommen. Dies trifft auf die Pfade $x_1, x_2, x_3, x_4, x_7, x_8$ und x_{10} zu. Die Belegungen durch diese sieben Pfade werden in das Flussnetz übertragen (Abb. 3.5a). Die Pfade x_6 und x_9 werden durch den berechneten LP-Wert 0.0 verworfen. In der Liste $OneZeroPaths$ ist nur 1 Pfad gespeichert, nämlich x_5. Es wird geprüft, ob der Pfad x_5 in die aktuelle Belegung des Flussnetzes passt. Dies ist aber nicht der Fall. Die Kante (b, c) kommt doppelt in diesem Pfad vor und überlastet somit die Kapazität dieser Kante (Abb. 3.5b).

Daher besteht die finale Pfadmenge auch aus den bislang genommenen Pfaden $x_1, x_2, x_3, x_4, x_7, x_8$ und x_{10}. Durch den heuristischen Ansatz wird in diesem Beispiel der Gesamtnutzwert von

$$71 \cdot 1 + 69 \cdot 1 + 48 \cdot 1 + 42 \cdot 1 + 83 \cdot 0 + 38 \cdot 0 + 55 \cdot 1 + 17 \cdot 1 + 7 \cdot 0 + 3 \cdot 1 = 305$$

berechnet und entspricht hierbei ca. 89% der Optimallösung.

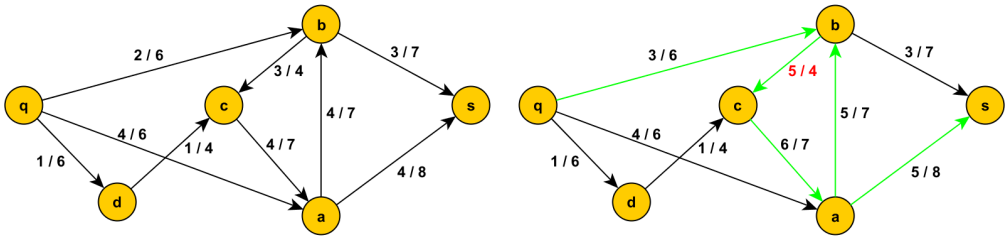

Abb. 3.5: Anwendung der LP-Heuristik

3.4.3 Laufzeitanalyse

Der erste aufwendige Schritt dieses Algorithmus (vgl. Algorithmus 3) ist das Finden einer relaxierten Lösung des LPs (Zeile 4). Wie schon in Kapitel 2.4.3 beschrieben, arbeitet der Simplex-Algorithmus im Worst Case in exponentieller Zeit. Dieser Nachteil taucht in der Praxis in der Regel nicht auf, sodass somit aus praktischen Gründen der Simplex-Algorithmus in der GlpK-Bibliothek verwendet wird. Das Ergebnis ist ein numerischer Wert für jede Variable x_i, der zwischen 0 und 1 liegt. Diese Pfade werden iterativ durchlaufen (Zeilen 5 bis 13) und überprüft, ob dieser LP-Wert 1 beträgt oder innerhalb des Intervalls $\{0, \ldots, 1\}$ liegt. Unter der Annahme, dass der Aufruf P.$get(i)$ linear verläuft, kann dieser Schleifenrumpf abgegrenzt werden mit:

$$\begin{aligned} & \mathcal{O}(p)\,(\mathcal{O}(p) + \mathcal{O}(p)) \\ = {} & \mathcal{O}(p)\,(2 \cdot \mathcal{O}(p)) \\ = {} & 2 \cdot \mathcal{O}(p^2) \\ \in {} & \mathcal{O}(p^2) \end{aligned}$$

Der nächste Schritt ist das Berechnen der aktuellen Flusswerte der bislang genommenen Pfade (Zeilen 14 bis 22). Die Laufzeit der dreifachen Schleife kann durch $\mathcal{O}(p \cdot pathlength_{max} \cdot m)$ beschrieben werden. Da wiederum ein Pfad Zyklen enthalten kann, ist die Anzahl an Kanten im Flussnetz abgrenzbar durch die Anzahl der Kanten im längsten Pfad $pathlength_{max}$. Daher folgt für diese Schleife:

$$\begin{aligned} & \mathcal{O}(p \cdot pathlength_{max} \cdot m) \\ = {} & \mathcal{O}(p \cdot pathlength_{max} \cdot pathlength_{max}) \\ = {} & \mathcal{O}(p \cdot pathlength_{max}^2). \end{aligned}$$

Das Sortieren der Pfade (Zeile 24) in der Liste $ZeroOnePaths$ wird erneut durch $\mathcal{O}(p \log p)$ ausgedrückt. Die nach dem LP-Wert sortierten Pfade werden absteigend durchlaufen und für jeden dieser Pfade wird geprüft, ob er noch in das Flussnetz passt. Da für die $Check\_Path$-Methode die Laufzeit in dieser Arbeit schon bewiesen worden ist (vgl. Kap. 3.1.3), kann die vorliegende Schleife (Zeilen 24 bis 28) durch $\mathcal{O}(p \cdot pathlength_{max}^2)$ beschrieben werden. Der letzte Schritt ist das Aufsummieren der Nutzwerte der genommenen Pfade (Zeilen 30 bis 32) und kann mit $\mathcal{O}(p)$ abgegrenzt werden.

Es wird daher die Ordnung des polynomiellen Anteils der Heuristik hergeleitet:

$$\begin{aligned} & \mathcal{O}(p^2 + p \cdot pathlength^2_{max} + p \cdot \log p + p\, pathlength^2_{max} + p) \\ = \; & \mathcal{O}(2 \cdot p \cdot pathlength^2_{max} + p^2 + p \, \log p + p) \\ \in \; & \mathcal{O}(p \cdot pathlength^2_{max}) \end{aligned}$$

Auch hierbei wird vorausgesetzt, dass ein Pfad Zyklen enthalten kann. Daher kann die Anzahl der Kanten in einem Pfad größer sein als die Gesamtanzahl an Kanten m im Flussnetz. Für eine azyklische Probleminstanz gilt die Ordnung: $\mathcal{O}(p\, m^2)$

Die Folgerung ist, dass der beschriebene Algorithmus mit Ausnahme des Simplex-Algorithmus in Polynomialzeit aufgeführt werden kann. Mit der Begründung, dass zwar in der Theorie ein exponentieller Aufwand hinsichtlich des Simplex-Algorithmus existiert, der aber praktisch nicht auftaucht, kann dennoch von einem polynomiellen Gesamtverhalten des Algorithmus ausgegangen werden. Die Erwartung der LP-Heuristik ist, dass hinsichtlich der Laufzeit eine deutlich bessere Messzeit für eine Probleminstanz resultiert als bei einem ganzzahligen LP basierend auf der binären Optimierung.

4 Evaluation der Ergebnisse

Dieses Kapitel beschäftigt sich im Allgemeinen mit der Erzeugung von Testdaten und den Ergebnissen der Testreihen. Zunächst wird die Path-Packing-Bibliothek vorgestellt, die für diese Abschlussarbeit angefertigt worden ist. Anschließend widmet sich dieses Kapitel der Erstellung von Flussnetzen und Pfadmengen, die mit unterschiedlichen Problemgrößen generiert werden können. Neben den erstellten Probleminstanzen sind die einzustellenden Parameter der Path-Packing-Bibliothek und der GlpK-Bibliothek von entscheidender Bedeutung. Die Ergebnisse in Form von Messzeiten, der Qualität als auch der Auslastung der Flussnetze werden daraufhin vorgestellt, ehe das definierte Modell auf seine Vor- und Nachteile untersucht wird.

4.1 Path-Packing-Library

Die algorithmischen Ansätze dieser Arbeit wurden auf einer Java basierenden Bibliothek implementiert und ausgewertet. Die Gesamtfunktionalität, der Aufbau der vorhandenen Datenstrukturen und die Anwendung dieser Bibliothek können im Anhang nachgeschlagen werden (vgl. Anhang A und B). Die beiden Greedy-Heuristiken sind als rein Java basierende Klassen umgesetzt worden. Für den Ansatz der Linearen Programmierung sowie der LP-Heuristik wurde ein Java-Wrapper (Schuchardt 2014) für die C basierende GlpK-Bibliothek verwendet. Neben den unterschiedlichen algorithmischen Ansätzen setzt diese Bibliothek auch das Erzeugen der Probleminstanzen um, fertigt die Messungen an und gibt diese arithmetisch aufbereitet aus.

4.2 Generierung der Testdaten

Um eine Auswertung der Ansätze durchführen zu können, müssen zunächst geeignete Testdaten erschaffen werden. Für die Generierung von Probleminstanzen werden zwei verschiedene Ansätze unterschieden. Einfach zu lösende Instanzen sind Flussnetze, die in der Regel eine einzige Engstelle besitzen. Schwere Probleminstanzen basierend auf Flussnetzen, bei denen möglichst viele Engstellen erzeugt werden, um die Komplexität dieser Instanz zu erhöhen. Auf Basis der erstellten Flussnetzen werden geeignete Pfadmengen generiert.

4.2.1 Generierung von einfachen Probleminstanzen

Dieses Unterkapitel beschäftigt sich mit dem Ansatz der einfachen Probleminstanzen und somit dem Erzeugen von Flussnetzen, deren Kantenkapazitäten auf randomisierten und untereinander unabhängigen Werten beruhen. Anschließend wird dann der algorithmische Ansatz vorgestellt, der die Pfadmenge, basierend auf dem erstellten Flussnetz generiert. Obwohl diese Art von Instanzen hierbei als „einfach" bezeichnet werden, kann es durch das randomisierte Vorgehen passieren, dass unter diesen einfachen Instanzen dennoch solche existieren, die schwer lösbar sind.

Erstellen von Flussnetzen

Für die Erstellung von einfachen Probleminstanzen wird der Algorithmus aus (Lau 2007, S. 27ff.) verwendet. Dieser erzeugt gewichtete Flussnetze, die auch zyklisch sein können. Die benötigten Parameter sind die Knotenanzahl n, die Kantenanzahl m, das minimale Kantengewicht $minweight$ und das maximale Kantengewicht $maxweight$. Knoten 1 wird als Quelle angesehen, Knoten n als Senke. Der anfängliche Ablauf sieht vor, dass Pfade von der Quelle zur Senke erstellt werden, bis entweder die benötigte Anzahl an Kanten erreicht ist oder sich alle Knoten auf einem direkten Pfad von der Quelle zur Senke befinden. Ist die zweite Bedingung erfüllt, so werden weitere Kanten erstellt bis die angegebene Anzahl m erreicht

ist. Die Kapazität einer Kante $c(e)$ ist eine randomisierte Ganzzahl zwischen den Grenzen $minweight$ und $maxweight$.

Für die Verwendung dieses Algorithmus im Kontext dieser Abschlussarbeit sind einige Modifikationen von Nöten, um diesen in die Path-Packing-Bibliothek zu integrieren:

- Die Eingabeparameter $seed$, $nodei$, $nodej$ und $weight$ werden entfernt.
- Knoten und Kanten werden direkt dem Graph-Objekt der Path-Packing-Bibliothek hinzugefügt, sodass die erzeugten Werte nicht in den vorgesehenen Arrays gespeichert werden müssen.
- Der neue Rückgabewert der Methode ist das erzeugte Graph-Objekt.

Die folgende Abbildung zeigt das Ergebnis eines Aufrufs dieser Methode mit den Parametern $n = 8$, $m = 12$, $minweight = 5.0$ und $maxweight = 15.0$. Durch die Knoten v_3, v_5 und v_6 zeigt der erzeugte Graph sogar ein zyklisches Verhalten.

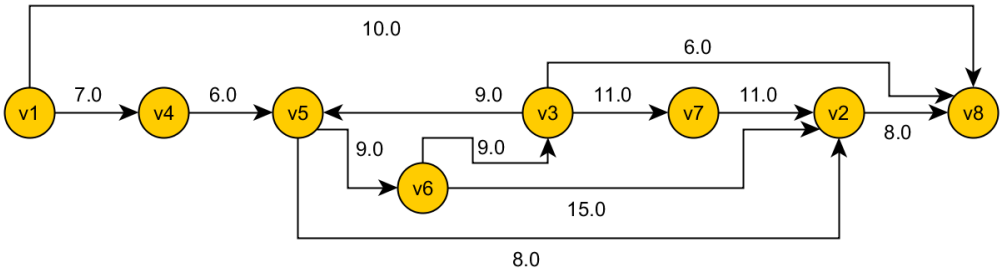

Abb. 4.1: Erstellen eines Flussnetzes einer einfachen Probleminstanz

Erstellen der Pfadmenge

Das Erzeugen von Pfaden geschieht auf Basis eines erzeugten Flussnetzes. Für diesen algorithmischen Ansatz (Alg. 4) werden drei Parameter benötigt: Die Anzahl der zu erzeugenden Pfade $nrpaths$, die Quelle q und die Senke s des Flussnetzes. Einem neuen Pfad-Objekt wird zunächst ein randomisierter numerischer Wert zugeordnet (Zeile 5), der als Nutzwert des Pfades dient. Die Variable $position$ zeigt auf die aktuelle gewählte Position des Pfades. Da jeder Pfad in der Quelle q anfängt, wird anfänglich der Variablen $position$ die Quelle q zugewiesen (Zeile 6). Ausgehend von der aktuellen Position werden alle Kanten in der Kantenmenge gesucht, deren Anfangsknoten dieser Position entspricht. Aus dieser Teilmenge der Kanten wird randomisiert eine Kante ausgewählt und dem Pfad hinzugefügt (Zeilen 7 bis 9).

Die beschriebenen Schritte werden solange ausgeführt, bis die aktuelle Position durch Zufall auf die Senke s des Flussnetzes trifft (Zeilen 10 bis 17). Dort endet jeder Pfad. Für jeden Pfad wird dann die Methode $calculateMaxCountEdge$ aufgerufen, welche die Anzahl der am meisten verwendeten Kanten in einem Pfad berechnet. Diese Anzahl ist nötig für die Berechnung des spezifischen Nutzwertes des Pfades. Letztendlich wird der Pfad zur Liste der erzeugten Pfade hinzugefügt (Zeile 19).

Durch die Anwendung dieses Algorithmus können auch Pfade erstellt werden, die Zyklen erhalten. Die Bedeutung und Konsequenz von Zyklen in einem Flussnetz ist schon in Kapitel 3.2 erwähnt worden.

```
1:  procedure CREATEPATHS(int nrpaths, String q, String s)
2:      paths ← NULL                                          ▷ Liste für erzeugte Pfade
3:      for (i = 0; i < nrpaths; i++) do
4:          path ← NULL
5:          path.setValue ← randomInt(100)                    ▷ Erzeuge randomisierten Nutzwert
6:          position ← q
7:          edges ← getEdgesWithSource(position)              ▷ Kanten, deren Quelle die aktuelle Position ist
8:          edge = edges.get(randomInt(edges.size()))         ▷ Wähle randomisiert eine nächste Kante
9:          path.addEdge(edge)                                ▷ Füge Kante zum Pfad hinzu
10:         while position ≠ sink do
11:             position = edge.getSink()                     ▷ Aktualisiere aktuelle Position
12:             edges = getEdgesWithSource(position)          ▷ Kanten, deren Quelle die aktuelle Position ist
13:             if position ≠ sink then
14:                 edge = edges.get(randomInt(edges.size())) ▷ Wähle randomisiert eine nächste Kante
15:                 path.addEdge(e)                           ▷ Füge Kante zum Pfad hinzu
16:             end if
17:         end while
18:         path.calculateMaxCountEdge()
19:         paths.add(path)
20:     end for
21: end procedure
```

Algorithmus 4: Methode zum Erzeugen von Pfaden für einfache Instanzen

4.2.2 Generierung von schwierigen Probleminstanzen

Die gemessenen Laufzeiten von Probleminstanzen eines Algorithmus sind ein wichtiger Faktor für den praktischen Einsatz. Diese hängt auch von der Schwierigkeit der erzeugten Probleminstanz ab. In dieser Abschlussarbeit wird daher bewusst ein Augenmerk auf die Erzeugung und Untersuchung schwieriger Instanzen gelegt. Die Frage nach einer akzeptablen Laufzeit ist nicht nur aus wissenschaftlicher, sondern vor allem aus praktischer Sicht ein interessanter Untersuchungspunkt. Im Zusammenhang mit dem Path-Packings zeichnen sich schwierige Probleminstanzen dadurch aus, dass ein Transportsystem viele Engstellen besitzt. Im übertragenen Sinne gilt es daher einen Ansatz festzulegen, um Flussnetze mit möglichst vielen minimalen Schnitten zu erstellen.

Erstellen des Flussnetzes

Ein Algorithmus für das Generieren schwieriger Probleminstanzen kann mit den folgenden Arbeitsschritten beschrieben werden:

1. Der erste Schritt ist das Erschaffen eines gerichteten azyklischen Graphen (engl. directed acyclic graph - DAG). Ein DAG mit n Knoten und m Kanten wird auf Basis des Algorithmus aus (Johnsonbaugh u. Kalin 1991, S. 153) erstellt. Hierbei werden die n Knoten mit den Kanten $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ verbunden, sodass ein zusammenhängender Graph entsteht. Dazu sind $n - 1$ Kanten nötig; um m Kanten zu erstellen, werden die fehlenden Kanten (v_i, v_j) durch randomisierte numerische Werte i und j mit der Eigenschaft $i < j$ erstellt. Durch das Sicherstellen, dass j echt größer als i ist, wird die Zykelfreiheit des Graphen garantiert.

2. Berechnung aller möglichen Pfade von der Quelle q zur Senke s; die Anzahl der Pfade wird mit der Variablen W bezeichnet. Eine eigene Umsetzung dieser Berechnung basierend auf dem Prinzip der Dynamischen Programmierung befindet sich im Anhang dieser Arbeit (vgl. Anhang D).

3. Berechnung aller Pfade von der Quelle q zur Senke s, die die Kante (v, w) beinhalten. Diese Anzahl von Pfaden wird für jede Kante (v, w) berechnet und mit der Variablen W_{vw} bezeichnet. Um die Variable W_{vw} für jede Kante zu berechnen, wird der vorherige Schritt ausgenutzt, da die Gesamtanzahl an Pfaden W nun bekannt ist. Aus dem Flussnetz wird die Kante (v, w) vorübergehend entfernt

und anschließend die Gesamtanzahl an Pfaden $W_{\neg vw}$ aus diesem neuen Flussnetz mit dem Algorithmus des zweiten Schritts berechnet. Diese Anzahl an Pfaden kann als Komplement von W_{vw} angesehen werden. Durch $W_{vw} = W - W_{\neg vw}$ wird so auf die Anzahl an Pfaden mit der Kante (v, w) geschlossen.

4. Berechnung des Wahrscheinlichkeitsquotienten $p_{vw} = \frac{W_{vw}}{W}$ für jede Kante (v, w). Dieser stellt den Anteil der Pfade über der Kante (v, w) im Verhältnis zu allen Pfaden dar.

5. Berechnung der Kapazität C_{cw} einer Kante. Der Wert p_{vw} wird mit einer festgelegten Musterkapazität C_{muster} multipliziert und ergibt so die berechnete Kantenkapazität. Zu diesem Zeitpunkt besitzen alle möglichen Schnitte des Flussnetzes denselben Wert und bilden so die schwerste aller Probleminstanzen für das erzeugte Flussnetz. Dies kann dadurch begründet werden, dass alle Schnitte des aktuellen Graphen gleichzeitig auch jeweils einen minimalen Schnitt bilden. Das Flussnetz besteht zu diesem Zeitpunkt nur aus Engpässen.

6. Runden von C_{vw} und Einrechnung einer Spanne (engl. range) für jede Kante. Da dieses Flussnetz die Realität so gut wie möglich abbilden soll, wird jede berechnete Kantenkapazität auf die nächste Ganzzahl gerundet und durch ein randomisiertes Einrechnen einer Spanne variabel gehalten. Die Schwierigkeit der Probleminstanz wird zwar dadurch ein wenig gelöst, bleibt aber trotzdem – im Vergleich zu den einfachen Instanzen – deutlich schwieriger. Falls bei der Einberechnung der Spanne negative Kantenkapazitäten entstehen, so werden diese auf die minimale ganzzahlige Kapazität von 1.0 gesetzt. Das Ziel ist somit ein Flussnetz mit möglichst vielen Schnitten zu erzeugen, die einem minimalen Schnitt entsprechen oder nahe an diesem liegen.

Erzeugen von Pfadmengen

Um schwierige Probleminstanzen zu erzeugen, muss ebenfalls gewährleistet sein, dass möglichst jeder Weg durch das Flussnetz auch durch Pfade in der Pfadmenge abgedeckt ist. Der Auswahlprozess für einen optimalen Algorithmus ist somit deutlich schwieriger, da die kombinatorischen Möglichkeiten, die zur Auswahl stehen, steigen. Daher ist die Auswahl auf Basis gleicher Wahrscheinlichkeiten (vgl. Kap. 4.2.1) von Nachteil. Das Flussnetz in Abbildung 4.2 zeigt eine solche nachteilige Situation.

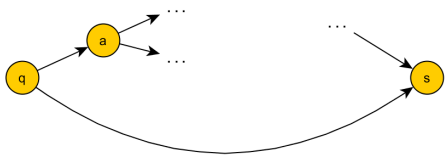

Abb. 4.2: Nachteil durch gleichwahrscheinliche Kantenwahl

In diesem Beispiel existiert ein direkter Pfad von der Quelle q zur Senke s. Alle anderen Pfade sind angedeutet, müssen aber über die Kante (q, a) gehen. Bei einer Pfadauswahl mit einer gleichwahrscheinlichen Auswahl würde in diesem Beispiel jede Kante, die aus der Quelle q herausgeht, mit der Wahrscheinlichkeit 0.5 ausgewählt werden. Dies hat zur Folge, dass jeder zweite Pfad in der endgültigen Pfadmenge statistisch gesehen dem Pfad $p =< (q, s) >$ entspricht. Somit kann nicht sichergestellt werden, dass möglichst alle Pfade dieses Flussnetzes erfasst werden können; insbesondere nicht die Pfade, die die Kante (q, a) enthalten. Je größer die Anzahl an unterschiedlichen Pfaden ist, desto kombinatorisch komplexer wird der Auswahlprozess für das Finden einer Optimallösung.

Um eine gleichmäßige Auslastung aller Kanten durch die Pfadmenge zu gewährleisten, werden die Pfade nicht auf Basis einer gleichverteilten Auswahlwahrscheinlichkeit gebildet, sondern auf einer Verteilungswahrscheinlichkeit basierend auf dem berechneten Wert p_{vw} einer Kante, der das Verhältnis der Pfade über (v, w) zu allen Pfaden darstellt.

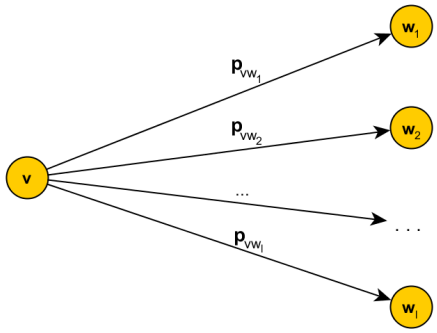

Abb. 4.3: Pfadauswahl auf Basis der p_{vw} Werte

Abbildung 4.3 zeigt eine allgemeine Entscheidungssituation innerhalb einer Pfadbildung. Von Knoten v soll der Folgeknoten bestimmt werden. Diese potentiellen Folgeknoten werden mit $w_1, w_2, \ldots, w_l$ bezeichnet. Die jeweiligen Kanten besitzen daher die Werte $p_{vw_1}, p_{vw_2}, \ldots, p_{vw_l}$. Daher lässt sich die Auswahlwahrscheinlichkeit für Knoten w_i wie folgt ausdrücken:

$$prob(w_i) = \frac{p_{vw_i}}{\sum_{k=1}^{l} p_{vw_k}}$$

4.3 Testdurchführung

Als Vorbereitung der Auswertung müssen einige Einstellungen in Bezug auf die Path-Packing- und Glpk-Bibliothek vorgenommen werden. Die Einstellungen dieser Parameter werden in den kommenden Abschnitten näher erläutert. Dabei werden zunächst die Hardware-Eigenschaften des verwendeten Rechners vorgestellt, anschließend die Parameter der Path-Packing-Bibliothek festgehalten, ehe Bezug auf die Auswertung der GlpK-Parameter genommen wird.

4.3.1 Technische Ressourcen

Die Messzeiten der erstellten Testreihen werden für einen einheitlichen Vergleich auf einer Maschine berechnet. Die technischen Eigenschaften dieser Maschine sind bewusst recht minimal gehalten, sodass ein Logistiklager auch mit solchen Minimalanforderungen die erstellten Probleminstanzen in den ermittelten Zeiten reproduzieren und berechnen kann. Logistiklager, die mit spezialisierten Maschinen ausgestattet sind, können somit die hier erstellten Messergebnisse als eine obere Zeitgrenze annehmen. Es ist offensichtlich, dass durch spezielle Rechenmaschinen oder gar Rechencluster bessere Laufzeiten erreicht werden. Die verwendete Maschine ist durch folgende technische Details charakterisiert:

- Mainboard: ASUSTeK P8Z77-V
- CPU: Intel Core i5-3570 @ 3.4GHz
- GPU: Nvidia GeForce GTX 550 Ti
- RAM: 8 GB Kingston DDR 3 Synchron 1333 MHz
- HD: 1TB Hitachi HDS72101

4.3.2 Testparametrisierung der Path-Packing-Bibliothek

Die Path-Packing-Bibliothek zeichnet sich dadurch aus, dass verschiedene Parameter für einen Testdurchlauf eingestellt werden können und anschließend automatisiert getestet werden. Für eine allgemeine Auswertung der Probleminstanzen werden für die Erzeugung unterschiedlicher Problemgrößen folgende Parameter festgelegt:

Problemgröße	Wert	Beschreibung
n	$\{100, \ldots, 2500\}$	Knotenanzahl im Flussnetz. n ist die allgemeine Problemgröße, auf die alle anderen Variablen bezogen werden.
m	$1.5\,n$	Kantenanzahl im Flussnetz.
p	$1.5\,n$	Anzahl der Pfade in der Pfadmenge P.
$minweight$	5	Minimale Kantenkapazität. Dieser Parameter wird nur für einfache Probleminstanzen verwendet.
$maxweight$	15	Maximale Kantenkapazität. Dieser Parameter wird nur für einfache Probleminstanzen verwendet.
C_{muster}	MinCut der einf. Instanz	Musterkapazität für ein Flussnetz. Dieser Parameter wird nur für schwierige Probleminstanzen verwendet.
$range$	$10\% \cdot C_{muster}$	Spanne für eine Kantenkapazität beträgt 10% der Musterkapazität.
$runs$	10	Anzahl der Testdurchläufe.
$pathruns$	5	Anzahl von unterschiedlichen Pfadmengen für ein gleichbleibendes Flussnetz.
$p.value$	$\{1, \ldots, 100\}$	Nutzwert eines Pfades gegeben als eine randomisierte Ganzzahl aus dem Intervall 1 bis 100.
ϵ	10^{-6}	Festgelegte Abweichung für den Ansatz des heuristischen LPs.

Tab. 4.1: Parametrisierung der Path-Packing-Bibliothek

Für eine festgelegte Problemgröße n werden somit 10 randomisierte Flussnetze generiert. Für jedes Flussnetz werden wiederum 5 verschiedene Pfadmengen erzeugt, aus denen dann durch die Anwendung der Algorithmen die besten Pfade hinsichtlich des Gesamtnutzwertes gepickt werden. Die Größe der Pfadmenge wird abhängig von der Problemgröße gewählt, so dass sie für jede Probleminstanz der Größe n hinreichend groß ist. Zumindest für einen planaren Graphen – und somit auch für ein planares Transportsystem – kann die Anzahl der Kanten mit der oberen Schranke $3\,n - 6$ abgegrenzt werden (vgl. Krumke u. Noltemeier 2005, S. 321). Die Anzahl der Kanten wird daher so festgelegt, dass dieser Parameter in der Mitte dieser Begrenzung liegt: $m = 1.5\,n$. Dadurch wird ein mittelmäßig verzweigtes Transportnetz repräsentiert.

Obwohl die Erzeugung von einfachen und schweren Instanzen unterschiedlich durchgeführt wird, soll bei den erzeugten Probleminstanzen darauf geachtet werden, dass sie von den Problemgrößen so ähnlich wie möglich sind, um so die Ergebnisse möglichst gut vergleichen zu können. Die Knotenanzahl n, die Kantenanzahl m, die Pfadanzahl p, die Anzahl der generierten Pfadmengen $pathruns$ sowie die Anzahl der Durchläufe $runs$ werden für die schwere Instanzen übernommen. Damit die Größe des minimalen Schnitts des Flussnetzes einer schwierigen Instanz möglichst der Größe einer einfachen Instanz ähnelt, wird die Musterkapazität C_{muster} auf die jeweilige berechnete Größe des minimalen Schnitts einer einfachen Instanz gesetzt. Das heißt, dass die berechneten minimalen Schnitte der einfachen Instanzen die Grundlage für die Größen der Engstellen der schwierigen Instanzen bilden. Durch das Runden auf die nächste Ganzzahl und das Einrechnen der Spanne variiert die Größe des minimalen Schnittes der schwierigen Instanz allerdings dennoch ein wenig.

4.3.3 Parameter des GlpK-Solvers

Die Path-Packing-Bibliothek verwendet für den Ansatz der Linearen Programmierung und der LP-Heuristik die GlpK-Bibliothek. Die GlpK-Bibliothek an sich bietet wiederum einige Einstellungsmöglichkeiten, die für die benötigte Messzeit einer Probleminstanz ausschlaggebend sind. Drei Parameter mit entscheidender Bedeutung für das Lösen von LPs sind der Branching-Algorithmus, der Back-Tracking-Algorithmus, der zuständig ist für das Durchsuchen des Verzweigungsbaumes und die Wahl des Schnittebenenverfahrens (vgl. GnuProject 2014d, S. 63f.).

Für den Branching-Algorithmus (engl. branching technique) stehen durch die GlpK-Bibliothek folgende Parameter zur Verfügung:

Abkürzung	GlpK-Parameter	Bedeutung
FFV	GLP_BR_FFV	First Fractional Variable. Hierbei ist die erste gebrochene Variable die Grundlage für das Branching.
LFV	GLP_BR_LFV	Last Fractional Variable. Das Branching wird auf die letzte gebrochene Variable angewandt.
MFV	GLP_BR_MFV	Most Fractional Variable. Die meist gebrochene Variable wird für das Branching verwendet. Damit ist die Variable gemeint, die am nächsten in der Mitte zweier Ganzzahlen liegt.
DTH	GLP_BR_DTH	Heuristic By Driebeck and Tomlin. Dies ist der voreingestellte Wert der GlpK-Bibliothek und basiert auf (Driebeck 1966; Tomlin 1970). Dabei basiert die Heuristik auf der Berechnung von Strafen für jede nicht erfüllte Ganzzahl. Auf Basis der größten Strafe wird das nächste Teilproblem ausgewählt.
PCH	GLP_BR_PCH	Hybrid Pseudo-Cost Heuristic. Die Branch-Variable wird auf Basis einer Abschätzung des Zielfunktionswertes ermittelt. Wenn dies nicht möglich ist, so wird das MFV-Verfahren angewendet (vgl. GnuProject 2014a, Z. 598ff.)

Tab. 4.2: Branching-Parameter der GlpK-Bibliothek

Für das Durchlaufen des Verzweigungsbaumes (engl. backtracking techniques) existieren folgende Parameter:

Abkürzung	GlpK-Parameter	Bedeutung
DFS	GLP_BT_DFS	Depth First Search. Der Verzweigungsbaum wird auf Basis einer Tiefensuche durchlaufen.
BFS	GLP_BT_BFS	Breadth First Search. Der Verzweigungsbaum wird auf Basis einer Breitensuche durchlaufen.
BLB	GLP_BT_BLB	Best Local Bound. Das nächste Teilproblem mit der besten lokalen Schranke wird als nächstes gelöst. Dies ist der voreingestellte Wert der GlpK-Bibliothek. (vgl. GnuProject 2014g, Z. 129ff.)
BPH	GLP_BT_BPH	Best Projection Heuristic. Das Teilproblem mit dem bestgeschätzten Zielfunktionswert wird gewählt. (vgl. GnuProject 2014f, Z. 99ff.)

Tab. 4.3: Back-Tracking-Parameter der GlpK-Bibliothek

Folgende Schnittebenenverfahren (engl. cut options) werden angeboten:

Abkürzung	GlpK-Parameter	Bedeutung
$NONE$	-	Alle Schnittebenenverfahren sind deaktiviert. Dies sind die voreingestellten Werte der GlpK-Bibliothek.
GMI	gmi_cuts	Gomory's Mixed Integer Cut Option
MIR	mir_cuts	Mixed Integer Rounding Cut Option
COV	cov_cuts	Mixed Cover Cut Option
CLQ	clq_cuts	Clique Cut Option

Tab. 4.4: Parameter der Schnittebenenverfahren der GlpK-Bibliothek

Diese drei Parameter werden zunächst auf kleine Problemgrößen angewandt. Die erreichten Messzeiten werden verglichen, um so eine möglichst gute Parametrisierung für die Testdurchführung der einfachen und schwierigen Probleminstanzen festzulegen. Die Entscheidung der Auswahl der endgültigen Parameter basiert auf den Problemgrößen schwieriger Probleminstanzen von $n = 300, 400$ und 500.

Auf Basis der Auswertung der drei Parameter (siehe Anhang E) wird festgehalten, dass folgende Parameter gewählt werden:

- Branching-Algorithmus : MFV-Verfahren.
- Back-Tracking-Algorithmus : BPH-Verfahren.
- Schnittebenenverfahren : CLQ-Verfahren.

Des Weiteren bietet die GlpK-Bibliothek einen Parameter für eine maximale Rechendauer einer einzelnen Probleminstanz an. Besonders für große Probleminstanzen kann die Dauer der Berechnung nicht abgesehen werden, sodass dieser Parameter für eine Probleminstanz auf die maximale Rechenzeit von 10 Stunden gesetzt wird. Dieses Vorgehen ist sinnvoll, da ein manueller Abbruch somit verhindert werden kann und die erstellte Probleminstanz immerhin noch für die heuristischen Ansätze berechnet wird.

4.4 Ergebnisse und Auswertung

Die erstellten Flussnetze und Pfadmengen bilden die Testdaten für die durchzuführenden Testdurchläufe. Diese werden, wie bei der Erstellung der Daten, in einfache und schwierige Instanzen gegliedert. Dabei liegt der Schwerpunkt der Auswertung zum einen auf den Messzeiten der Durchläufe und zum anderen auf der erreichten Qualität in Bezug zur Optimallösung. Auch die Auslastung des Flussnetzes in Form des erreichten Flusswertes des minimalen Schnitts soll dabei betrachtet werden.

4.4.1 Einfache Instanzen

Aufgetragen in Abbildung 4.4 sind die Messzeiten der vier beschriebenen Ansätze für einfache Instanzen. Dabei sind die Problemgrößen $n = 100$ bis 2500 in je 100er-Schritten gemessen worden. Zu erkennen ist eindeutig, dass die Greedy-Heuristik (blau) und die Greedy-Heuristik mit dem spezifischen Nutzwert (rot) durchgängig die besten Laufzeiten zeigen. Die Verläufe der beiden Greedy-Heuristiken sind nahezu identisch. Deswegen wird die Greedy-Heuristik mit dem spezifischen Nutzwert gestrichelt dargestellt, um den Verlauf der ursprünglichen Greedy-Heuristik ebenfalls erkennen zu können. Die praktischen Messzeiten decken sich folglich auch durch die Ermittlung der theoretischen Laufzeit (vgl. Kap. 3.1.3 und 3.2.3), da das verwendete Sortierkriterium der einzige Unterschied zwischen den beiden Greedy-Heuristiken ist. Die LP-Heuristik (grün) zeigt eine Laufzeit, die im Vergleich zu den Greedy-Heuristiken nur ein wenig schlechter ist. Diese Verschlechterung ist allerdings nur bei den größten gemessenen Problemen zu erkennen. Die exponentielle Laufzeit zur Ermittlung der optimalen Lösung durch die ganzzahlige Lineare Programmierung (gelb) weist mit deutlichem Abstand die schlechtesten Messzeiten vor. Für eine einzelne Probleminstanz der Größe $n = 2500$ werden im arithmetischen Mittel ca. 40 Minuten gebraucht. Im Vergleich dazu benötigt die LP-Heuristik für dieselben Probleminstanzen im Mittel nur 70 Sekunden.

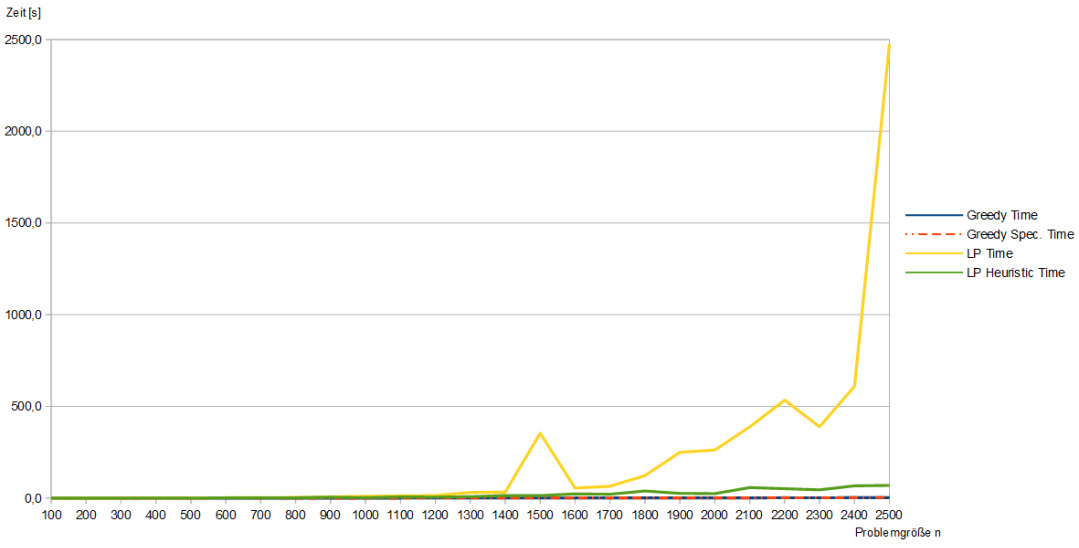

Abb. 4.4: Messzeiten der einfachen Instanzen

Es lässt sich feststellen, dass bzgl. der Messzeit die drei Heuristiken eindeutig ein besseres Ergebnis liefern als das ganzzahlige LP. Je größer die Problemgröße gewählt wird, desto eher kommt dabei der Nachteil der exponentiellen Laufzeit zum Tragen.

Die Frage ist allerdings nicht nur, ob eine Verbesserung der Laufzeit erreicht worden ist, sondern auch, wie hoch der qualitative Verlust der heuristischen Ansätze in Bezug zum LP ist. Das Ergebnis des ganzzahligen LPs ist die Optimallösung. Dies kann durch keinen weiteren Ansatz übertroffen werden und wird daher für alle Problemgrößen mit 100% gleichgesetzt.

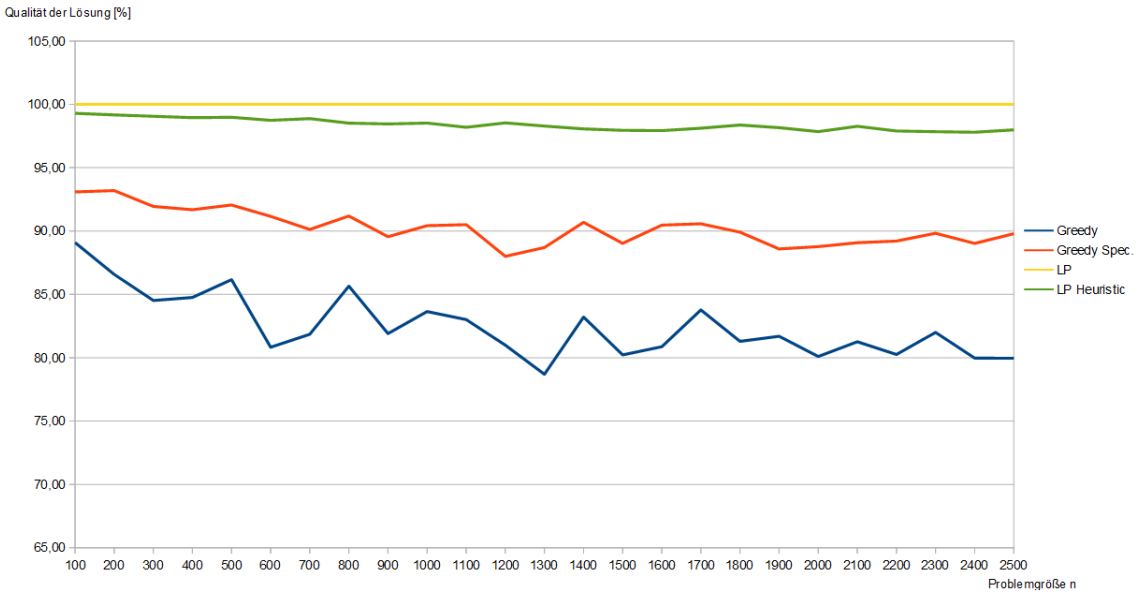

Abb. 4.5: Qualität der einfachen Instanzen

Abbildung 4.5 zeigt den prozentualen Unterschied zwischen der Optimallösung und den Lösungen der einzelnen Heuristiken. Die Qualität ist hierbei definiert als der erreichte Gesamtnutzwert der Heuristiken in Bezug zum optimalen Gesamtnutzwert des ganzzahligen LPs. Während die beiden Greedy-Heuristiken die beste Laufzeit besitzen, kann nur eine unzureichende Qualität für diese Heuristiken festgestellt werden. Die Greedy-Heuristik erreicht über alle Instanzen hinweg durchschnittlich 82% im Vergleich zur Optimallösung. Die Greedy-Heuristik mit dem spezifischen Nutzwert schneidet besser ab und erreicht im Durchschnitt 90%. Allerdings liegt der Ansatz des heuristischen LPs mit durchschnittlich 98% deutlich am nächsten an der Optimallösung. Ebenfalls ist zu erkennen, dass alle vier Verfahren nur einen sehr geringen Qualitätsverlust für wachsende Probleminstanzen aufweisen, sodass zumindest von einem fast konstanten Verhalten der Qualität ab einer Problemgröße von $n = 1000$ gesprochen werden kann.

Der minimale Schnitt ist die Engstelle eines Flussnetzes. Die aktuelle Auslastung dieser Engstelle ist ein Faktor für das Potential um weitere Kisten auf das Transportband schicken zu können. Abbildung 4.6 belegt diese Auslastung. Der minimale Schnitt wird als Optimum dargestellt (violett), welches der maximalen Auslastung und somit 100 % in der jeweiligen Probleminstanz entspricht. Die Fragestellung ist, wie nah die algorithmischen Ansätze diesem Optimum kommen. Das ganzzahlige LP zeigt eindeutig das beste Verhalten. Es wird deutlich, dass sich dieses Verfahren vor allem bei größer werdenden Instanzen dem Optimum nähert. Dicht dahinter folgt die LP-Heuristik, danach die beiden Greedy-Heuristiken. Diese Reihenfolge deckt auch das Ergebnis hinsichtlich der Qualität des Nutzwertes. Auffällig ist allerdings auch, dass alle vier Verfahren für kleine Problemgrößen ($n = 100$ bis 400) ein schlechteres Verhalten zeigen als bei größeren Instanzen.

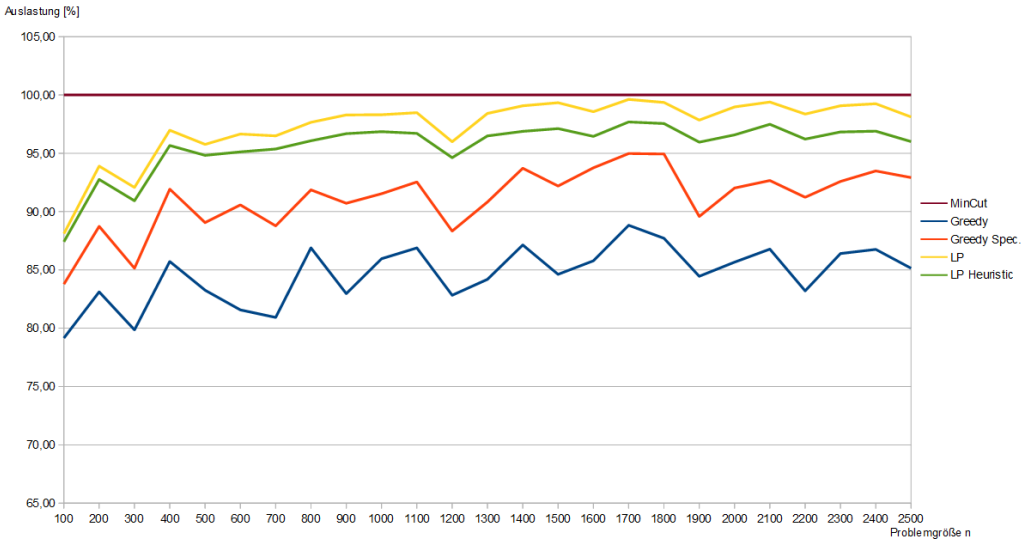

Abb. 4.6: Auslastung des minimalen Schnitts

Zusammenfassend zeigt die LP-Heuristik für die einfachen Probleminstanzen das beste Verhalten. Zum einen greift der Vorteil des nachgewiesenen polynomiellen Verhaltens der Laufzeit, zum anderen werden eine sehr gute Qualität und Auslastung des minimalen Schnittes für die Probleminstanzen gezeigt. Die Greedy-Heuristiken besitzen zwar die beste Laufzeit, erreichen aber im Vergleich mit der LP-Heuristik eine geringere qualitative Leistung. Im Gegensatz dazu ist die hohe Laufzeit der große Nachteil des ganzzahligen LPs. Für Problemgrößen, die noch größer werden, ist dies nicht mehr rentabel. Der Vorteil ist jedoch, dass die Lösung eines ganzzahligen LPs grundsätzlich der Optimallösung für eine Probleminstanz entspricht und somit mit keinem anderen Verfahren überboten werden kann.

4.4.2 Schwierige Instanzen

Die interessante Frage ist, ob und in welchem Ausmaß Abweichungen bei den Messzeiten, der Qualität und der Auslastung schwerer Instanzen auftreten. Da die Zykelfreiheit der schweren Instanzen garantiert ist, gilt bei jedem Flussnetz, dass der Nutzwert gleich dem spezifischen Nutzwert ist. Daher muss die Greedy-Heuristik auf Basis des spezifischen Nutzwertes nicht weiter betrachtet werden.

Abbildung 4.7 zeigt die Messzeiten der schweren Instanzen. Aus zeitlichen Gründen sind nur Instanzen der Größen 100 bis 700 berechnet worden. Es ist zu erkennen, dass die Messzeit des ganzzahligen LPs (gelb) schon bei einer Problemgröße $n = 700$ ein exponentiell wachsendes Verhalten zeigt, welches bei den einfachen Instanzen erst bei einer Problemgröße von $n = 2500$ auftritt. Die durchschnittliche Berechnungszeit beträgt ca. $14440s \approx 4h$ für eine schwere Instanz dieser Größe. Im Vergleich dazu beträgt die Messzeit der Problemgröße $n = 700$ bei den einfachen Instanzen für die optimale Lösung im Mittel nur ca. 3 Sekunden. Die Greedy-Heuristik (blau) sowie das heuristische LP (grün) brauchen für die berechneten schweren Instanzen erneut nur wenige Sekunden. Um beide Verläufe erkennen zu können, wird die LP-Heuristik gestrichelt dargestellt.

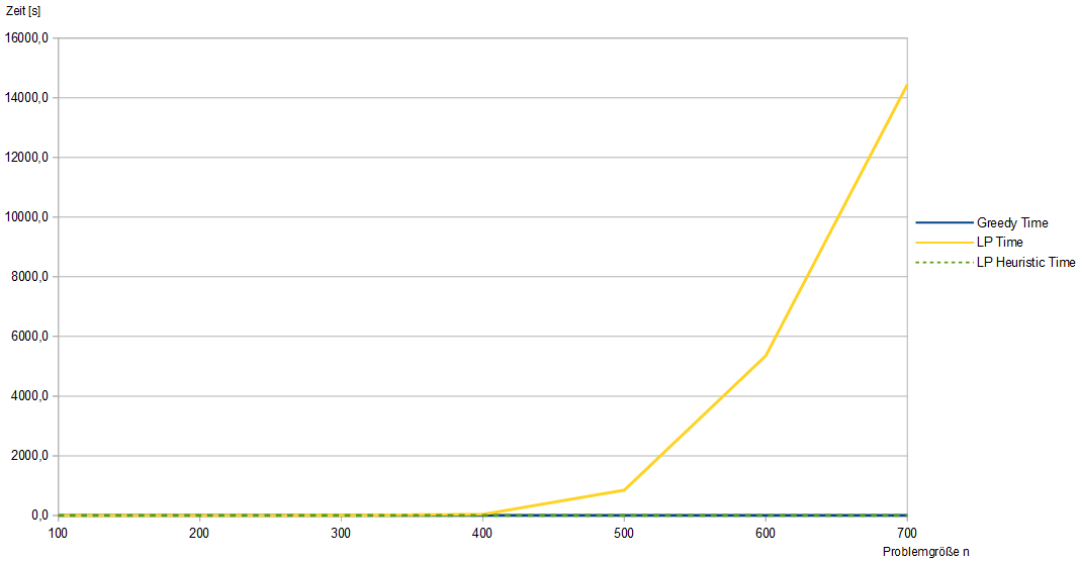

Abb. 4.7: Messzeiten der schweren Instanzen

Auch für diese Probleminstanzen wird die erreichte Qualität betrachtet. Abbildung 4.8 zeigt die prozentualen Unterschiede zwischen der Optimallösung und den beiden Heuristiken der schweren Instanzen. Die LP-Heuristik zeigt ein deutlich besseres Verhalten als die Greedy-Heuristik und liegt erneut knapp unterhalb der Optimallösung. Sowohl für die Greedy-Heuristik als auch bei der LP-Heuristik ist allerdings eine leichte Verschlechterung der qualitativen Leistung bei größer werdenden Instanzen sichtbar. Für die Problemgröße $n = 700$ ist bei den schweren Instanzen im arithmetischen Mittel nur noch ein qualitativer Wert von $95,13\%$ zu verzeichnen; im Vergleich dazu liegt dieser Wert bei den einfachen Instanzen bei $98,87\%$. Die Messreihe des heuristischen LPs zeigt ein annähernd lineares abnehmendes Verhalten. Dieses ist sogar noch deutlicher bei der Greedy-Heuristik ausgeprägt. Dort wird bei der Problemgröße $n = 600$ die Grenze von 80% klar unterschritten. Bei den einfachen Probleminstanzen zeigt die Kurve der Greedy-Heuristik, selbst bis zu einer Problemgröße von $n = 2500$, ein „asymptotisches" Verhalten gegen diese 80%-Grenze. Die Frage ist, ob dieses Verhalten ebenfalls bei den schweren Instanzen für größere Probleminstanzen eintritt oder ob die qualitative Leistung weiterhin abnimmt. Die Untersuchung größerer Probleminstanzen ist daher notwendig.

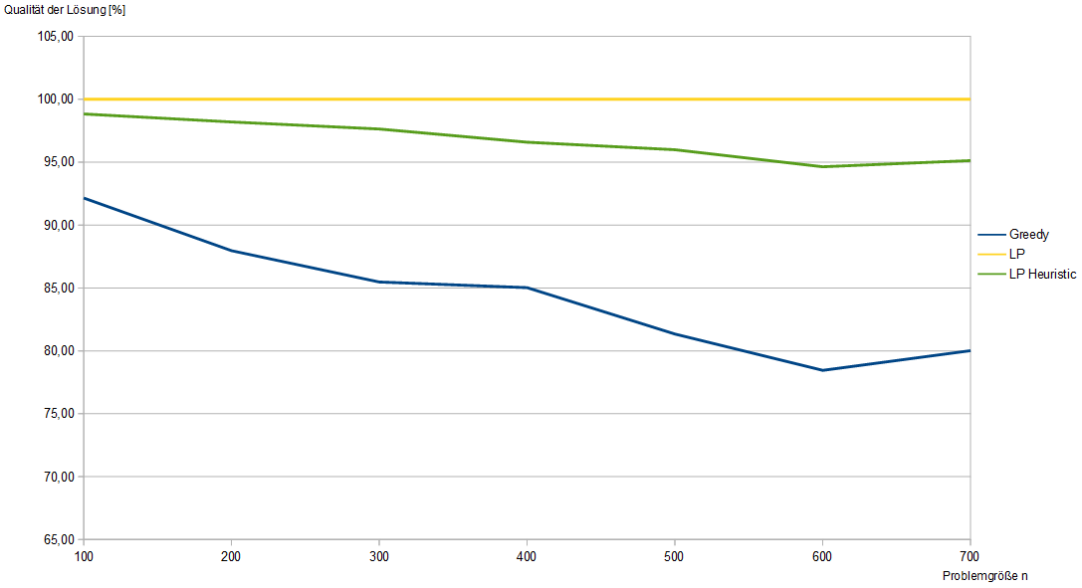

Abb. 4.8: Qualität der schweren Instanzen

Die Betrachtung der genauen Messzeiten zeigt allerdings auch, dass die gemessenen Zeiten bei einer gleichbleibenden schweren Problemgröße sehr stark variieren. Bspw. kann für die Problemgröße $n = 700$ unter den gemessenen Probleminstanzen festgehalten werden, dass die minimale Messzeit nur $61s$ beträgt; die maximale Messzeit kann mit über $10h$ festgehalten werden, da dies der eingestellte Parameter für die maximale Messzeit der GlpK-Bibliothek ist.

Einige Auffälligkeiten existieren ebenfalls bei der Betrachtung der Auslastung des minimalen Schnittes. Während die Optimallösung bei den einfachen Probleminstanzen in der Regel nahezu dem Wert des minimalen Schnittes entspricht, zeigen sich bei den schweren Instanzen deutliche Abweichungen. Die Optimallösung erreicht bei allen Problemgrößen nur noch eine Auslastung zwischen 83 bis 88%. Das heuristische LP liegt knapp dahinter in einem Bereich von 80 bis 86%. Deutlich abgeschlagen ist die Greedy-Heuristik. Sie beginnt knapp unter einer Auslastung von 80%, nimmt sogar bei größeren Probleminstanzen deutlich ab. Bei einer Größe der Probleminstanz von $n = 700$ liegt die Auslastung nur noch bei ca. 66%.

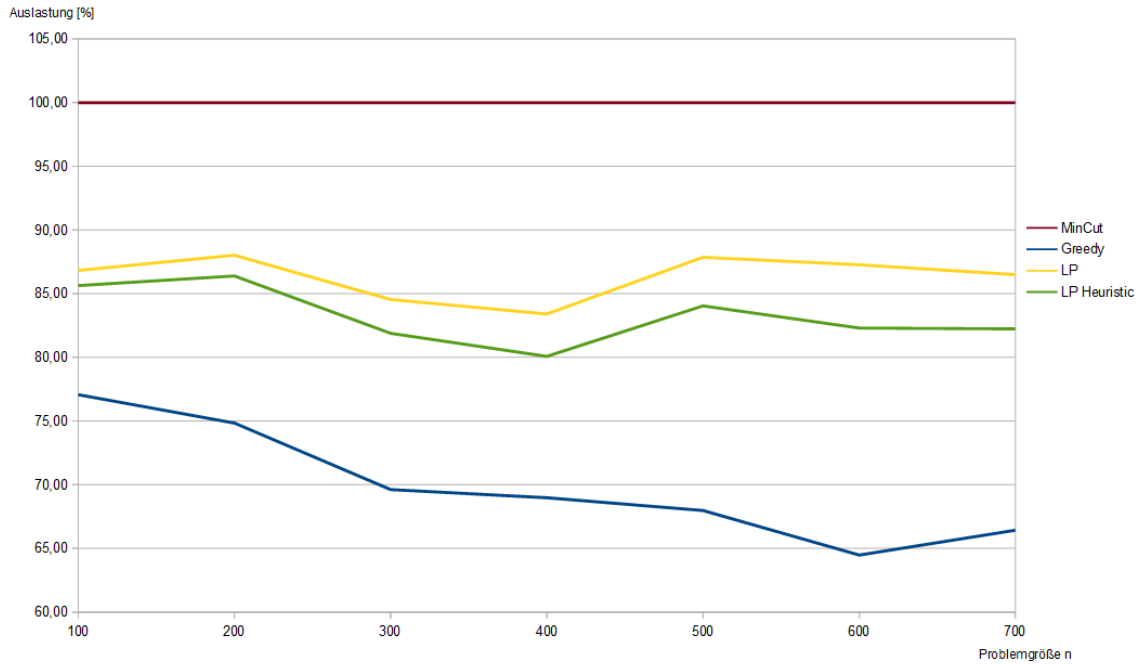

Abb. 4.9: Auslastung des minimalen Schnitts

Es kann somit gezeigt werden, dass die Lösungen von schweren Probleminstanzen ein teilweise schlechteres Verhalten zeigen als bei den einfachen Instanzen. Dennoch kann hier erneut die LP-Heuristik als das Verfahren herausgestellt werden, welches das beste Gesamtverhalten zeigt. Bei diesem liegt die Laufzeit aller berechneten Probleminstanzen weiterhin bei nur wenigen Sekunden. Sowohl die qualitativen Ergebnisse als auch die Auslastung des Flussnetzes zeigen ein Ergebnis, welches zumindest nahe an die Optimallösung herankommt. Die Greedy-Heuristik zeigt erneut in allen drei Untersuchungskriterien die schlechteste Leistung. Schon bei den einfachen Instanzen ist der Nachteil des ganzzahligen LPs die Laufzeit; bei den schweren Instanzen kommt dieser Nachteil zum einen früher und zum anderen sogar noch gravierender zum Vorschein.

4.5 Diskussion des Modells

Dieses definierte Modell kann als abstrakte Vereinfachung eines Logistiklagers betrachtet werden. Das vorgeschriebene Ziel, Blockaden und Staus auf den Transportbahnen zu vermeiden, ist erfüllt worden. Bei den Greedy-Heuristiken wird kein Pfad genommen, der im übertragenen Sinne eine Kante überlastet. Diese

Bedingung wird ebenfalls beim Ansatz der Linearen Programmierung in Form der Nebenbedingungen erfüllt. Dieses Ziel ist eine der Grundvoraussetzungen für einen praktischen Einsatz des Modells. Garantiert ist auch, dass jede Kiste das Logistiklager an der Senke verlässt. Der Verlust einer Kiste ist laut dieser Modellierung ausgeschlossen.

Als positiv kann ebenfalls die Existenz eines mathematischen Verfahrens, nämlich der LP-Heuristik, bewertet werden, welches sowohl in seiner Laufzeit als auch in der Qualität und der Auslastung des Flussnetzes punktet. Insbesondere gilt die Laufzeit einer Berechnung als wichtiger Faktor für die Praxis, die für diesen heuristischen Ansatz bei einfachen und schwierigen Instanzen für alle berechneten Probleminstanzen im Sekundenbereich liegt. Die Anzahl der Knoten im Flussnetz repräsentiert die Zwischenstationen eines Logistiklagers. Die Problemgröße von $n = 700$ repräsentiert somit eine enorme Anzahl von Zwischenstationen, sodass auch von den berechneten Problemgrößen her gesehen das Modell als praxistauglich angesehen werden kann.

Allerdings sind einige Schwachstellen der Modellierung zu erwähnen. Das Flussnetz bildet zwar das Transportsystem eines Lagers ab, allerdings nur solche, die sehr einfach strukturiert sind. Es wird bislang angenommen, dass Kisten nur über eine einzige Quelle in das System hineingelangen und dieses nur über eine einzige Senke wieder verlassen. Dies muss für eine Weiterentwicklung des Modells modifiziert werden. Auch die Modellierung der Arbeitsplätze – auch Bahnhöfe genannt – ist nur unzureichend modelliert. In der Praxis wird eine Kiste auf eine separate Transportschiene umgelenkt, wenn diese zum Arbeitsplatz eines Mitarbeiters gelangen muss. Das bisherige modellierte Flussnetz enthält zwischen zwei Knoten nur eine einzige Kante, die sowohl die Transportbahn an sich als auch die separate Transportschiene des Bahnhofs repräsentiert. Durch eine detailliertere Modellierung können folglich wiederum mehr Kisten transportiert werden, da für eine Übergangszeit von wenigen Minuten einige Kisten in dieser separaten Transportschiene zwischengelagert werden, ehe diese auf das Transportband zurückgegeben werden.

Der größte Schwachpunkt der Modellierung ist allerdings die Umsetzung des zeitlichen Aspektes. Eine Kiste beitritt das System an der Quelle und wird dann entlang eines bestimmten Pfades transportiert. Dabei nimmt das Modell an, dass diese Kiste bis zum Verlassen an der Senke, zu jedem Zeitpunkt jede Kante ihres Pfades blockiert. Dies ist natürlich in der Praxis nicht der Fall. Wenn eine Kiste über eine Zwischenstation weitergereicht wird, wird die Kante, von der diese Kiste kommt, wieder frei und kann so mit anderen Kisten für diesen Zeitpunkt belastet werden. Es folgt automatisch eine noch höhere Dichte an transportierten Kisten in dem Transportnetz. Das Modell muss demzufolge um eine Zeitkomponente erweitert werden, die eine Differenzierung der Auslastung von unterschiedlichen Zeitpunkten realisiert.

Die theoretische Annahme, dass die gewählten Kisten zeitgleich auf das Band gegeben werden und abgearbeitet werden müssen, bevor eine weitere Kistenmenge – quasi der nächste Durchlauf – auf das Transportsystem gegeben wird, ist nicht realitätsgetreu abgebildet. Die Betrachtung der Auslastung unterschiedlicher Zeitpunkte und das dynamische Hineingeben weiterer Kisten wird auch als Verstetigung des Modells bezeichnet, welche als notwendige praktische Eigenschaft eines realen Modells gilt.

Weiterhin muss auch das Transportieren von Kisten an den Zwischenstationen untersucht werden. Auch dort können zu einem Zeitpunkt zwei Kisten bislang kollidieren. Diese Kollision muss durch eine geschickte Modellierung verhindert werden, indem sich zu keinem Zeitpunkt die Wege zweier Kisten in dieser Zwischenstation kreuzen.

5 Zusammenfassung und Ausblick

Dieses Kapitel fasst die Hauptziele dieser Arbeit zunächst zusammen, die somit einen Überblick über die geleistete Arbeit geben. Anschließend werden einige Gedanken aufgezählt, die auf die zukünftige Erweiterung des Modells abzielen.

5.1 Zusammenfassung

Diese Abschlussarbeit hat das Ziel, die Frage des Path-Packings innerhalb der Lagerlogistik zu untersuchen. Die Hauptaufgabe ist die Erschaffung eines graphentheoretischen Modells, welches durch vier verschiedene Lösungsansätze analysiert wurde. Diese Ansätze sind zwei Greedy-Heuristiken, die Berechnung der Optimallösung durch ein ganzzahliges LP sowie die LP-Heuristik, welche die Vorteile der Laufzeit und einer guten Qualität kombiniert.

Nebenbei gibt es Tätigkeiten, die zur Umsetzung der oben genannten Verfahren erfüllt werden müssen. Diese sind die Erzeugung von randomisierten Flussnetzen in Form von einfachen und schwierigen Probleminstanzen, den dazugehörigen Pfadmengen und der Ausgabe und Aufbereitung der Ergebnisse.

Eine vereinfachte, aber geeignete Darstellung der Lagerstrukturen ist die Realisierung als Flussnetz. Die Kantenkapazitäten bilden in dieser Modellierung die obere Grenze für die Anzahl an Kisten, die über eine bestimmte Kante transportiert werden können. Auf Grundlage dieser Kapazitäten wählen die mathematischen Heuristiken und Algorithmen die Pfade aus. Die beiden Greedy-Heuristiken überzeugen zwar mit ihrer Laufzeit, können allerdings nur eine mittelmäßige qualitative Leistung erbringen. Das gegenteilige Verhalten zeigt das ganzzahlige LP, welches eine exponentielle Laufzeit besitzt, aber folglich auch die Optimallösung liefert. Die beiden Vorteile werden durch die LP-Heuristik erfüllt. Diese stellt sich somit als das Verfahren mit dem besten Gesamtverhalten heraus.

Die Differenzierung zwischen einfachen und schweren Probleminstanzen zeigt ebenfalls, dass die LP-Heuristik auch komplexe Testfälle lösen kann. In den Vordergrund rückt dabei allerdings die exponentielle Laufzeit des ganzzahligen LPs, deren Auswirkungen schon bei kleinen Problemgrößen sichtbar werden.

Das definierte Modell zeigt allerdings auch Schwächen, da die Anpassung der Modellierung an reale Datenmengen, wie bspw. die Modellierung der Bahnhöfe, die variable Anzahl an Aufträgen und die Herleitung des Nutzwertes, nicht gegeben ist. Auch eine zeitliche Komponente ist nicht im Modell integriert worden.

Das Resultat der Abschlussarbeit ist somit ein Modell, welches eine gute Grundlage für weitere Untersuchungen bietet. Die LP-Heuristik kann aus den beschriebenen Gründen als eine Möglichkeit angesehen werden die besten Pfade für einen möglichst guten Gesamtnutzwert zu erhalten. Das zukünftige Potential ist eindeutig gegeben, um die aktuelle Modellierung zu erweitern und einen praktischen Einsatz nahe zu bringen.

5.2 Ausblick

Das definierte Modell bietet großes Potential für Erweiterungen. Die im vorherigen Kapitel (vgl. Kap. 4.5) diskutierten Nachteile des Modells, nämlich die Abbildung auf reale Lagerstrukturen und der zeitlichen Verstetigung sind zwei wesentliche Aspekte, die die weitere Untersuchung des Path-Packings erfordern.

Die Flussnetze werden innerhalb der Abschlussarbeit auf randomisierten Testparametern generiert. Für eine weitere Untersuchung ist die Erzeugung und Verwendung realistischer Datenmengen nötig. Die Anzahl an

Kanten eines erzeugten Flussnetzes wird bislang nur durch die Planaritätsbedingung festgelegt. Ein Abgleich von realen Verzweigungen innerhalb eines Lagers findet nicht statt. Die Erzeugung realer Strukturen beinhaltet vor allem die Festlegung der Anzahl an Zwischenstationen, die Verzweigungsdichte an Transportbahnen und die Modellierung der Arbeitsplätze der Kommissionierer.

Ebenfalls müssen die Anzahl der Kisten und die Eigenschaften eines Pfades auf Basis realer Datenmengen abgebildet werden. Für diese Arbeit basiert die Anzahl der Pfade auf der jeweiligen Problemgröße, um eine immer hinreichend große Pfadmenge zur Verfügung zu haben. Innerhalb eines Jahres kann die Anzahl an Aufträgen pro Tag allerdings stark variieren, sodass mit steigenden Auftragszahlen auch die Komplexität der Berechnung steigt. Die weiteren Untersuchungsaspekte sind daher die Auswirkungen von Änderungen dieser Parameter auf die Laufzeit, die Qualität und die Auslastung eines Flussnetzes.

Alle vier mathematischen Ansätze versuchen einen möglichst guten Gesamtnutzwert zu berechnen. Diese Arbeit stellt einen Nutzwert als randomisierten numerischen Wert dar, geht aber nicht weiter auf die Herleitung dieses Nutzwertes ein. Je höher ein Nutzwert ist, desto prioritärer muss eine Kiste abgearbeitet werden. Diese Priorität kann auf unterschiedlichen Faktoren beruhen, wie bspw. die Lieferzeit von Lkws, unterschiedlichen Kundenklassen oder bestimmten Eigenschaften der Waren innerhalb dieser Kiste. Eine argumentative fundierte Abbildung dieser Faktoren auf den genannten numerischen Wert ist ebenfalls von Nöten.

Um ein solches Modell in der Praxis anwenden zu können, ist es von Vorteil auch ungeplante Vorkommnisse innerhalb eines Lagers in den aktuellen Auswahlprozess der Kisten einzuberechnen. Vorkommnisse, wie bspw. ein technischer Defekt an einem Transportband, das zu Bruch gehen einer Ware oder die Abwesenheit eines Kommissionierers an seinem zuordneten Bahnhof zwingen den Auswahlprozess zu spontanen Änderungen bei der Auswahl der Kisten. Die Überlastung von einzelnen Transportbahnen bzw. den anzusteuernden Bahnhöfen gilt es so zu verhindern. Ein geeigneter Algorithmus muss daher flexibel und dynamisch agieren, um Vorkommnisse innerhalb der Lagerstruktur auszugleichen und Blockaden somit vorzubeugen.

Literaturverzeichnis

(Brucker 2007) BRUCKER, P.: *Scheduling Algorithms*. 5. Auflage. Berlin : Springer Verlag, 2007

(Cormen u. a. 2001) CORMEN, T. ; LEISERSON, C. ; RIVEST, R. ; STEIN, C.: *Introduction to Algorithms*. 2. Auflage. Cambridge : The MIT Press, 2001

(Driebeck 1966) DRIEBECK, N.: An algorithm for the solution of mixed-integer programming problems. In: *Management Science* 12 (1966), Nr. 1, S. 576–587

(Ehrhardt+Partner 2015a) EHRHARDT+PARTNER: *Lagerführungssystem LFS - Die integrierte Gesamtlösung*. Koblenz, 2015. – [Stand vom 05.06.2015]

(Ehrhardt+Partner 2015b) EHRHARDT+PARTNER: *Warehouse Logistics*. `http://www.ehrhardt-partner.com/warehouse-management/`, 2015. – [Stand vom 31.05.2015]

(Garey u. Johnson 1979) GAREY, M. ; JOHNSON, D.: *Computers and Intractability - A Guide to the Theory of NP-Completeness*. New York : W. H. Freeman and Company, 1979

(GnuProject 2014a) GNUPROJECT: *Choose Branching Variable With Pseudocost Branching*. `file://glpk-4.55/src/glpios09.c`, 2014

(GnuProject 2014b) GNUPROJECT: *Clique Cut Generator*. `file://glpk-4.55/src/glpios08.c`, 2014

(GnuProject 2014c) GNUPROJECT: *Cover Inequalities*. `file://glpk-4.55/src/glpios07.c`, 2014

(GnuProject 2014d) GNUPROJECT: *GNU Linear Programming Kit for GLPK Version 4.45*. `file://glpk-4.55/doc/glpk.pdf`, 2014

(GnuProject 2014e) GNUPROJECT: *Most Fractional Variable*. `file://glpk-4.55/src/glpios09.c`, 2014

(GnuProject 2014f) GNUPROJECT: *Select Subproblem Using The Best Projection Heuristic*. `file://glpk-4.55/src/glpios12.c`, 2014

(GnuProject 2014g) GNUPROJECT: *Select Subproblem With Best Local Bound*. `file://glpk-4.55/src/glpios12.c`, 2014

(GnuProject 2014h) GNUPROJECT: *The structure CFG describes the conflict graph.* `file://glpk-4.55/src/cglib/cfg.h`, 2014

(Heinrich 2014) HEINRICH, M.: *Transport- und Lagerlogistik*. 9. Auflage. Hamburg : Springer Vieweg, 2014

(Jarre u. Stoer 2004) JARRE, F. ; STOER, J.: *Optimierung*. Berlin : Springer Verlag, 2004

(Johnsonbaugh u. Kalin 1991) JOHNSONBAUGH, R. ; KALIN, M.: A Graph Generation Software Package. In: *SIGCSE 91 Proceedings of the twenty-second SIGCSE technical symposium on Computer science education* 23 (1991), Nr. 1, S. 151–154

(Karp 1972) KARP, R.: Reducibility Among Combinatorial Problems. In: *R. E. Miller and J. W. Thatcher, Complexity of Computer Computations (Plenum Press)* (1972), S. 85–103

(Khachiyan 1980) KHACHIYAN, L.: Polynomial Algorithms In Linear Programming. In: *U.S.S.R. Comput. Maths. Math. Phys. (Plenum Press)* 20 (1980), Nr. 1, S. 53–72

(Koop u. Moock 2008) KOOP, A. ; MOOCK, H.: *Lineare Optimierung*. Berlin : Spektrum Akademischer Verlag, 2008

(Krumke u. Noltemeier 2005) KRUMKE, S. ; NOLTEMEIER, H.: *Graphentheoretische Konzepte und Algorithmen*. Wiesbaden : Teubner-Verlag, 2005

(Lau 2007) LAU, H.: *A Java Library Of Graph Algorithms and Optimization*. Boca Raton : Chapman and Hall CRC, 2007

(Makhorin 2012) MAKHORIN, A.: *GLPK (GNU Linear Programming Kit)*. `https://www.gnu.org/software/glpk/`, 2012. – [Stand vom 31.05.2015]

(Martello u. Toth 1990) MARTELLO, S. ; TOTH, P.: *Knapsack Problems*. Chichester : John Wiley and Sons, 1990

(Naveh 2015) NAVEH, B.: *JGraphT*. `http://jgrapht.org/`, 2015. – [Stand vom 31.05.2015]

(Nickel u. a. 2011) NICKEL, S. ; STEIN, O. ; WALDMANN, K.: *Operations Research*. Heidelberg : Springer, 2011

(Östergard 2002) ÖSTERGARD, P.: A fast algorithm for the maximum clique problem. In: *Discrete Applied Mathematics* 120 (2002), Nr. 1, S. 197–207

(Ottmann u. Widmayer 2012) OTTMANN, T. ; WIDMAYER, P.: *Algorithmen und Datenstrukturen*. 5. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2012

(Schuchardt 2014) SCHUCHARDT, H.: *GLPK for Java*. `http://glpk-java.sourceforge.net/`, 2014. – [Stand vom 31.05.2015]

(Sedgewick 2005) SEDGEWICK, R.: *Algorithmen in C*. München : Pearson Studium, 2005

(Syslo u. a. 2007) SYSLO, M. ; DEO, M. ; KOWALIK, J.: *Discrete Optimization Algorithms*. Mineola : Dover Publications Inc., 2007

(Tomlin 1970) TOMLIN, J.: Branch and bound methods for integer and non-convex programming. In: *in J. Abadie (ed.) Integer and Nonlinear Programming* (1970), S. 437–450

(Turau 2009) TURAU, V.: *Algorithmische Graphentheorie*. 3. Auflage. München : Oldenbourg Verlag, 2009

(Wegener 2003) WEGENER, I.: *Komplexitätstheorie*. Berlin : Springer-Verlag, 2003

(yWorks 2015) YWORKS: *yWorks - the diagramming company*. `http://www.yworks.com/en/products/yfiles/yed/`, 2015. – [Stand vom 04.06.2015]

(Zimmermann u. Stache 2001) ZIMMERMANN, W. ; STACHE, U.: *Operations Research*. München : Oldenbourg Wissenschaftsverlag GmbH, 2001

Anhang

Anhang A - Dokumentation der Path-Packing-Bibliothek

Anhang A stellt die allgemeine Dokumentation der Path-Packing-Bibliothek dar. Dabei werden die Funktionen der Bibliothek beschrieben, die Beziehungen der Klassen untereinander anhand eines UML-Diagramms dargestellt und anschließend die Anwendung der Bibliothek auf Basis von Beispielen vorgeführt.

Allgemeine Beschreibung

Die Path-Packing-Bibliothek bietet die Funktionalität, verschiedene Probleminstanzen der definierten Ansätze auszuwerten. Die beiden Greedy-Heuristiken (Kap. 3.1 und 3.2), der Ansatz der Linearen Programmierung (Kap. 3.3) sowie die LP-Heuristik (Kap. 3.4) sind dabei in dieser Java basierenden Bibliothek implementiert worden. Angeboten werden ebenfalls Methoden für das Erstellen von Flussnetzen einfacherer und schwieriger Probleminstanzen sowie dazu passenden Pfadmengen (Kap. 4.2.1 und 4.2.2). Die automatisierte Testdurchführung kann durch spezielle Klassen, die die Aufrufe der Algorithmen steuern, durchgeführt werden.

Für das Berechnen des minimalen Schnittes eines Graphen ist die JGraphT-Bibliothek (Naveh 2015) angebunden worden. Es existiert eine Adapter-Methode, um eine Graphinstanz der Path-Packing-Bibliothek in eine Graphinstanz der JGraphT-Bibliothek zu transformieren, um die Berechnung des minimalen Schnitts dort durchzuführen.

Für eine tabellarische und strukturierte Ausgabe von berechneten Werten ist eine eigene Bibliothek namens *ConsoleTable* entwickelt worden (siehe Anhang C).

Klassenstruktur

Im Folgenden werden die Hauptklassen dieser Bibliothek mit den jeweils wichtigsten Methoden kurz und übersichtlich vorgestellt:

- GreedyAlgorithm.java : Diese Klasse beinhaltet die beschriebene Greedy-Heuristik und die dazugehörigen Funktionen. Durch das Aufrufen der Methode *setSortCategory()* kann zwischen der ursprünglichen Greedy-Heuristik und der Greedy-Heuristik basierend auf dem spezifischen Nutzwert unterschieden werden. Die öffentliche Methode *calculate()* kann für den Aufruf einer einmaligen Probleminstanz verwendet werden. Durch das Anbinden der JGraphT-Library können die Kanten, die den minimalen Schnitt bilden, berechnet werden; dies geschieht durch den Aufruf der Methode *calculateMinCutEdges()*. Durch Getter- und Setter-Methoden können alle verwendeten Variablen gesetzt und erhalten werden.

- LPAlgorithm.java : Der Ansatz der Linearen Programmierung ist in dieser Klasse implementiert. Die Grundfunktionen dieser Klasse sind identisch mit der Klasse der Greedy-Heuristik. Allerdings wird hierbei die optimale Lösung durch eine Anbindung des erwähnten Java-Wrappers und der GlpK-Bibliothek berechnet. Diverse Methoden erstellen die benötigte Zielfunktion, die Variablen sowie die Nebenbedingungen des LP. Diese werden dem LPObject der GlpK-Bibliothek übergeben.

- LPHeuristicAlgorithm.java : Diese Klasse implementiert die kombinierte Lösung der Linearen Programmierung und dem heuristischen Ansatz. Vom Aufbau ist diese Klasse fast identisch mit der des LP-Algorithmus. Die Variablen des LPs sind allerdings als kontinuierlich umgesetzt worden. Die *checkPath()*-Methode wurde von der Greedy-Heuristik übernommen und prüft die Pfade, die einen berechneten LP-Wert zwischen 0 und 1 haben.

- AlgorithmInterface.java : Die drei erwähnten Algorithmenklassen implementieren dieses Interface. Neben den gemeinsamen Methoden, die die Klassen zu implementieren haben, werden zwei Aufzählungstypen, nämlich *SortCategory* und *PrintCategory*, definiert. Die dort festgelegten Konstanten sind nötig für das Sortieren nach bestimmten Werten der Pfadmengen sowie für die Ausgabeart der berechneten Daten.

- GraphGenerator.java : Die beiden Ansätze zum Erstellen von einfachen und schwierigen Probleminstanzen werden in dieser Klasse implementiert. Die Methoden *randomDAG()* und *randomFlownet()* erzeugen die benötigten Flussnetzinstanzen.

- AlgorithmRunnerSimpleInstances.java : Diese Klasse bietet die Möglichkeit, automatisiert einfache Testinstanzen erzeugen und berechnen zu können. Da alle Algorithmen durch die Schnittstelle *AlgorithmInterface* einheitliche Methoden besitzen, können diese durch eine Schleife effektiv aufgerufen werden.

- AlgorithmRunnerHardInstances.java : Diese Klasse ist identisch zur vorherigen, allerdings erstellt diese schwere Probleminstanzen und dazugehörige Pfadmengen.

- Flownet.java : Ein Flussnetz besitzt eine statische Variable, die das Graph-Objekt speichert. Die Hauptfunktion dieser Klassen ist das Erstellen der Pfadmengen durch die Methoden *createPaths()* bzw. *createPathsByProbability()*. Weiterhin kann durch den Aufruf von *getMinCutEdges()* der minimale Schnitt des gespeicherten Graph-Objektes berechnet werden. Nützlich ist ebenfalls die Möglichkeit des Exportierens des Graph-Objektes in ein XML-Format, sodass der erzeugte Graph durch andere Software-Werkzeuge, wie bspw. yEd (yWorks 2015), visualisiert werden kann. Dies wird durch den Aufruf der Methode *saveGraphXML()* realisiert.

- Graph.java : Ein Graph-Objekt beinhaltet nur die für den Kontext der Path-Packing-Bibliothek benötigten Operationen. Dieses Objekt speichert Knoten, die durch Strings repräsentiert werden, und Kanten, die durch die Klasse *Edge.java* dargestellt werden. Die Methode *getJGraphT()* transformiert die gegebene Graphenstruktur in die Struktur für die angebundene JGraphT-Bibliothek.

- Edge.java : Eine Kante wird durch zwei Knoten ausgedrückt und kann unterschiedliche zugeordnete Werte speichern, wie bspw. die Kapazität, den W_{vw}- und p_{vw}-Wert und den aktuellen Flusswert der Kante.

- Path.java : Ein Pfad wird durch eine Instanz dieser Klasse repräsentiert und besteht aus einer Folge von Kanten. Wichtige Methoden sind *maxCountEdge()*, die die maximale Anzahl von identischen Kanten in diesem Pfad ermittelt und die Methode *calculateSpecValue()*, die den spezifischen Nutzwert dieses Pfades berechnet.

Klassenbeziehungen

Das UML-Klassendiagramm (Abb. 5.1) zeigt die Beziehungen der Klassen untereinander. Aus Gründen der Übersicht wird auf die Darstellung von Getter- und Setter-Methoden sowie auf die dazugehörigen Testklassen verzichtet.

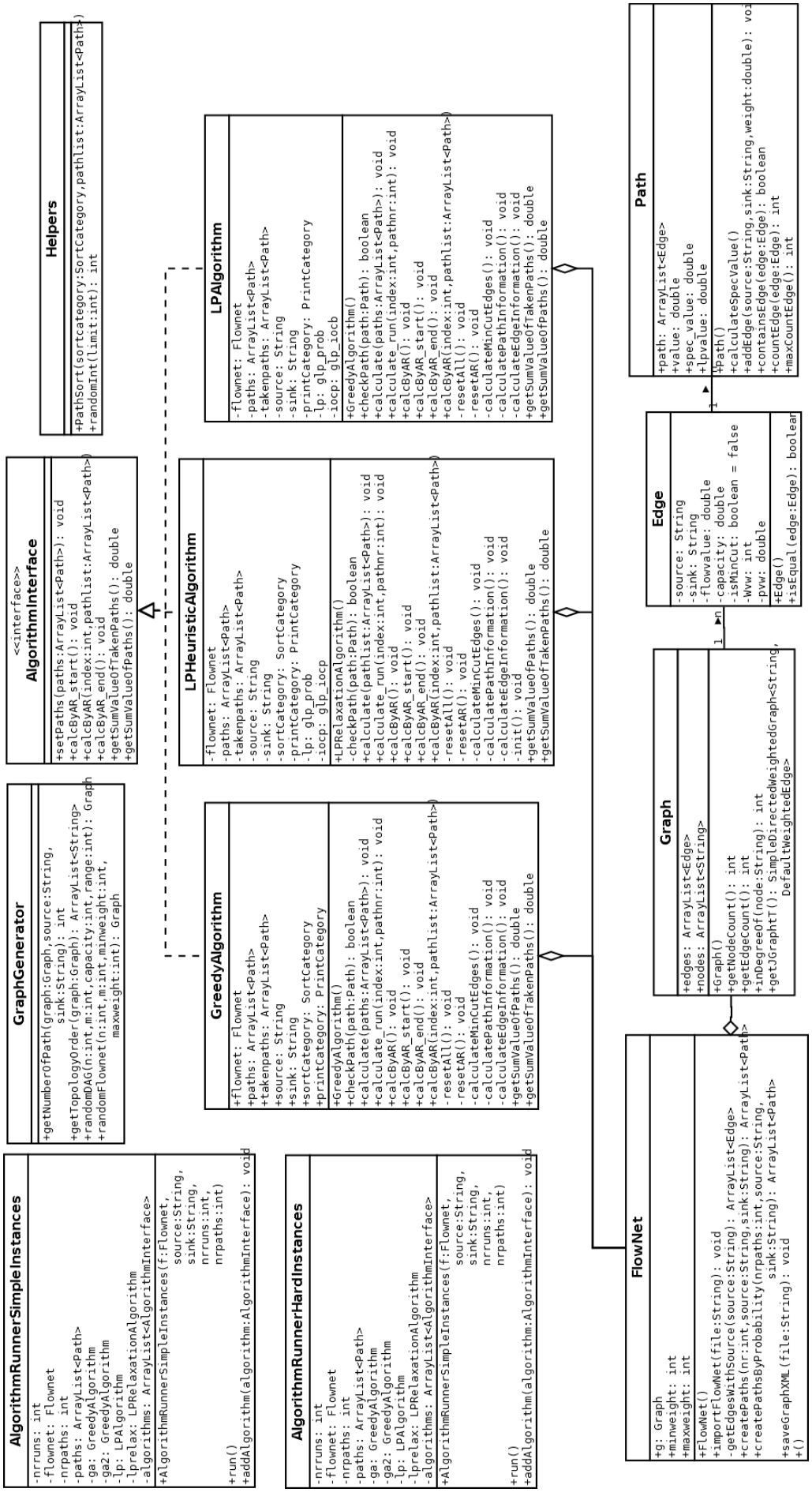

Abb. 5.1: UML-Klassendiagramm der Path-Packing-Bibliothek

Anhang B - Anwendungsszenarien der Path-Packing-Bibliothek

Um die praktische Anwendung dieser Bibliothek zu demonstrieren, werden im Folgenden einige Anwendungsszenarien dargestellt. Diese zeigen die Grundfunktionalitäten, um so die exemplarische Durchführung zu verdeutlichen, die für die Ergebnisse der einfachen und schwierigen Instanzen angewandt worden sind.

Anwendungsbeispiel 1

Das folgende Beispiel (Alg. 5) zeigt die Anwendung des Graph-Objektes. Dabei werden Knoten und Kanten mit ihren Kapazitäten hinzugefügt. Der minimale Schnitt dieses Graphen wird anschließend durch die angebundene JGraphT-Bibliothek berechnet und als Ausgabe auf der Standardkonsole ausgegeben.

```
public static void main(String[] args) throws IOException {

    //New graph object
    Graph g = new Graph();

    //Add nodes
    g.addNode("q");
    g.addNode("a");
    g.addNode("b");
    g.addNode("c");
    g.addNode("d");
    g.addNode("s");

    //Add edges
    g.addEdge("q", "a", 3.0);
    g.addEdge("q", "c", 2.0);
    g.addEdge("a", "b", 2.0);
    g.addEdge("a", "s", 1.0);
    g.addEdge("b", "c", 1.0);
    g.addEdge("c", "s", 4.0);
    g.addEdge("q", "d", 3.0);
    g.addEdge("d", "b", 2.0);
    g.addEdge("d", "a", 1.0);

     //Calculate min cut
    MinSourceSinkCut<String, DefaultWeightedEdge> mincut =
       new MinSourceSinkCut<String, DefaultWeightedEdge>( g.getJGraphT() );
    mincut.computeMinCut("q", "s");

    System.out.println("MinCut: " + mincut.getCutWeight());
    System.out.println("Source-Partition: " + mincut.getSourcePartition());
    System.out.println("Sink-Partition: " + mincut.getSinkPartition());
    System.out.println("MinCut-Edges: " + mincut.getCutEdges());
}
```

Algorithmus 5: Verwendung des Graph-Objektes und Berechnung des minimalen Schnitts

Anwendungsbeispiel 2

Das folgende Beispiel (Alg. 6) demonstriert die Anwendung der Bibliothek bzgl. des AlgorithmRunners für einfache Probleminstanzen, wie sie für die automatisierten Testreihen verwendet wird. Dabei wird die Problemgröße n von 100 bis 500 in 100er-Schritten erhöht. Für jede Problemgröße n werden 5 Pfadmengen generiert. Der Aufruf der *run()*-Methode des AlgorithmRunner-Objektes startet die Gesamtberechnung und gibt die Ergebnisse der Berechnungen auf der Standardkonsole aus. Die *randomFlownet()*-Methode erzeugt ein randomisiertes Flussnetz auf Basis des vorgestellten Algorithmus aus Kapitel 4.2.1.

```
public static void main(String[] args) throws IOException {
  // number of nodes
  int i = 500;

  for (int n = 100; n <= i; n = n + 100) {

    // create test parameters.
    int m = (int) (1.5 * n);
    int minweight = 5;
    int maxweight = 15;
    int path_nr = (int) (1.5 * n);
    int path_runs = 5;

    String source = "n1";
    String sink = "n" + String.valueOf(n);

    // Create a graph object
    Graph g = GraphGenerator.randomFlownet(n, m, minweight, maxweight);
    FlowNet flownet = new FlowNet(g);
    flownet.setEdgeCapacities(minweight, maxweight);

    // Create AlgorithmRunner for simple instances
    AlgorithmRunnerSimpleInstances ar = new
      AlgorithmRunnerSimpleInstances(flownet, source, sink, path_runs, path_nr);
    ar.run();
  }
}
```

Algorithmus 6: Beispiel für die Anwendung des AlgorithmRunners

Das Ergebnis dieser Berechnungen sind Tabellen mit den gemessenen und berechneten Werten für den jeweiligen Durchlauf. Dabei sind die ausgegebenen Parameter in Tabelle 5.1 näher erläutert:

Parameter	**Bedeutung**
$\#$	Die Nummer des Durchlaufes.
$\#n$	Die Knotenanzahl des aktuellen Flussnetzes bzw. die Problemgröße der aktuellen Probleminstanz.
$\#m$	Die Kantenanzahl des aktuellen Flussnetzes.
$min \quad c(e)$	Die minimale Kantenkapazität.
$max \quad c(e)$	Die maximale Kantenkapazität.
$\#p$	Die Anzahl der Pfade der aktuellen Pfadmenge.
$GREEDY$	Die Summe der Nutzwerte der genommen Pfade für die Greedy-Heuristik. Äquivalent dazu existieren die Parameter GR_SPEC, LP und $LP - RELAX$ für die anderen Ansätze.
MAX	Der kumulierte Nutzwert aller Pfade der aktuellen Pfadmenge.
$t[ms]$	Die Messzeit einer Probleminstanz unter Verwendung des jeweiligen Algorithmus. Die Zeit wird in Millisekunden angegeben.
$CurrentMC$	Die aktuelle Auslastung des minimalen Schnitts.
MC	Der Wert des minimalen Schnitts.

Tab. 5.1: Ausgabeparameter der Path-Packing-Bibliothek

Die folgende Abbildung zeigt einen Ausschnitt der Ausgabe des beschriebenen Beispiels. Es ist der Ausschnitt für die Problemgröße $n = 500$:

#	#n	#m	min c(e)	max c(e)	#p	GREEDY	MAX	t[ms]	Current MC	MC
1	500	750	5	15	750	2481.0	37079.0	287.0	26.0	33.0
2	500	750	5	15	750	2608.0	37288.0	102.0	28.0	33.0
3	500	750	5	15	750	2516.0	37591.0	264.0	26.0	33.0
4	500	750	5	15	750	2634.0	37375.0	142.0	30.0	33.0
5	500	750	5	15	750	2542.0	37652.0	264.0	27.0	33.0
						2556.2	37397.0	211.8	27.4	33.0

#	#n	#m	min c(e)	max c(e)	#p	GR_SPEC	MAX	t[ms]	Current MC	MC
1	500	750	5	15	750	2704.0	37079.0	266.0	29.0	33.0
2	500	750	5	15	750	3017.0	37288.0	110.0	33.0	33.0
3	500	750	5	15	750	2590.0	37591.0	235.0	27.0	33.0
4	500	750	5	15	750	2713.0	37375.0	297.0	29.0	33.0
5	500	750	5	15	750	2769.0	37652.0	172.0	30.0	33.0
						2758.6	37397.0	216.0	29.6	33.0

#	#n	#m	min c(e)	max c(e)	#p	LP	MAX	t[ms]	Current MC	MC
1	500	750	5	15	750	3086.0	37079.0	1719.0	33.0	33.0
2	500	750	5	15	750	3128.0	37288.0	2047.0	33.0	33.0
3	500	750	5	15	750	3133.0	37591.0	1563.0	33.0	33.0
4	500	750	5	15	750	3093.0	37375.0	1531.0	33.0	33.0
5	500	750	5	15	750	3104.0	37652.0	1500.0	33.0	33.0
						3108.8	37397.0	1672.0	33.0	33.0

#	#n	#m	min c(e)	max c(e)	#p	LP-RELAX	MAX	t[ms]	Current MC	MC
1	500	750	5	15	750	2997.0	37079.0	1594.0	32.0	33.0
2	500	750	5	15	750	3038.0	37288.0	1642.0	32.0	33.0
3	500	750	5	15	750	3133.0	37591.0	1519.0	33.0	33.0
4	500	750	5	15	750	3093.0	37375.0	1548.0	33.0	33.0
5	500	750	5	15	750	3104.0	37652.0	1532.0	33.0	33.0
						3073.0	37397.0	1567.0	32.6	33.0

Abb. 5.2: Ausschnitt der Ausgabe des Anwendungsbeispiels

Anwendungsbeispiel 3

Das Anwendungsbeispiel (Alg. 7) legt dar, wie ein Flussnetz randomisiert erstellt und anschließend als XML-Datei gespeichert wird. Diese Exportfunktion ist die Grundlage für eine anschließende Visualisierung durch andere Software-Programme, wie bspw. yEd.

```
public static void main(String[] args) throws IOException {

    // create test parameters
    int n = 10;
    int m = (int) (1.5 * n);
    int minweight = 5;
    int maxweight = 15;

    String source = "n1";
    String sink = "n" + String.valueOf(n);

    // Create a graph object
    Graph g = GraphGenerator.randomFlownet(n, m, minweight, maxweight);
    FlowNet flownet = new FlowNet(g);
    flownet.setEdgeCapacities(minweight, maxweight);

    flownet.saveGraphXML("graphfile.graphml");

}
```

Algorithmus 7: Demonstration der Exportfunktion der Path-Packing-Bibliothek

Nachteilhaft ist allerdings, dass für eine geeignete Visualisierung alle Knoten des abgespeicherten Graphen noch keine Koordinaten besitzen, um diese übersichtlich in einer Ebene darzustellen. Das Programm yEd interpretiert dies so als ob alle Knoten übereinander liegen würden. Abhilfe schafft der Aufruf des Flowchart-Algorithmus, welcher durch das Software-Werkzeug yEd angeboten wird. Dieser ordnet die gegebenen Knoten so an, dass diese nicht mehr übereinander angezeigt werden. Die Quelle und die Senke des Flussnetzes dienen dabei als die äußeren positionierten Knoten. Die gerichteten Kanten bilden so einen übersichtlichen Fluss, der – zumindest für eine geringe Knotenanzahl – sehr gut nachvollzogen werden kann. Abbildung 5.3 zeigt die Anwendung des Flowchart-Algorithmus für ein Flussnetz mit 10 Knoten.

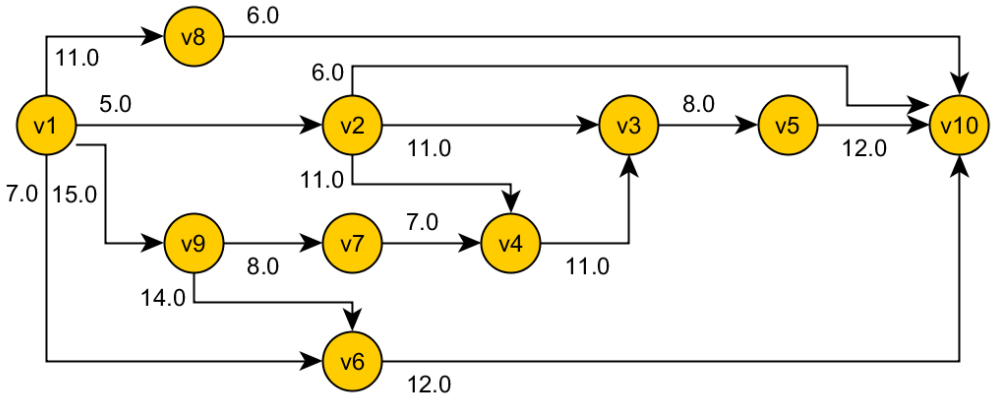

Abb. 5.3: Visualisierung eines Graphen durch das XML-Format in yEd

Anhang C - ConsoleTable-Library

Durch das Berechnen von Probleminstanzen entsteht eine große Menge an numerischen Ergebnissen. Um diese interpretieren zu können, müssen diese strukturiert ausgegeben werden. Die *ConsoleTable*-Bibliothek liefert einen Funktionsumfang, der dieser Anforderung gerecht wird. Die *Table*-Klasse speichert diese Werte in tabellarischer Form und gibt diese auf der Konsole aus. Alternativ können diese Werte auch im CSV-Format ausgegeben werden, um diese bspw. in Tabellenkalkulationsprogrammen effektiv weiterverarbeiten zu können. Der folgende Quellcode (Alg. 8) zeigt ein kurzes Beispiel für die Benutzung dieser Bibliothek:

```
public class Test {
  public static void main(String[] args) {
    Table table = new Table(4);
    Row row1 = new Row();
    row1.add(new SData("Title 1"));
    row1.add(new SData("Title 2"));
    row1.add(new SData("Title 3"));
    row1.add(new SData("Title 4"));
    table.add(row1);

    Row row2 = new Row();
    row2.add(new DData(1111.0));
    row2.add(new DData(222222.0));
    row2.add(new DData(33333333.0));
    row2.add(new DData(1234567890.0));
    table.add(row2);

    Row row3 = new Row();
    row3.add(new SData("Stringvalue 1"));
    row3.add(new SData("VeryLongStringvalue 2 "));
    row3.add(new SData("Value 3"));
    row3.add(new SData("Value 4"));
    table.add(row3);

    Row row4 = new Row();
    row4.add(new IData(12345));
    row4.add(new IData(12345));
    row4.add(new IData(12345));
    row4.add(new IData(12345));
    table.add(row4);

    table.print();
    table.printCSV();
  }
}
```

Algorithmus 8: Beispiel für die Anwendung der ConsoleTable-Bibliothek

Das Ergebnis ist in der nächsten Abbildung zu sehen. Zum einen werden die gespeicherten Werte in tabellarischer Form (Abb. 5.4a) dargestellt; zum anderen im CSV-Format (Abb. 5.4b).

```
-------------------------------------------------------------------
|      Title 1 |              Title 2 |     Title 3 |      Title 4 |
-------------------------------------------------------------------
|       1111.0 |             222222.0 | 3.3333333E7 | 1.23456789E9 |
| Stringvalue 1 | VeryLongStringvalue 2  |     Value 3 |      Value 4 |
|         12345 |                12345 |       12345 |        12345 |
-------------------------------------------------------------------
```

(a)

```
Title 1 , Title 2 , Title 3 , Title 4
1111.0 , 222222.0 , 3.3333333E7 , 1.23456789E9
Stringvalue 1 , VeryLongStringvalue 2 , Value 3 , Value 4
12345 , 12345 , 12345 , 12345
```

(b)

Abb. 5.4: Ausgabe der ConsoleTable-Bibliothek

Anhang D - Berechnung der Pfadanzahl eines DAGs

Für das Erzeugen von schweren Instanzen muss, wie in Kapitel 4.2.2 beschrieben, die Gesamtanzahl W an Pfaden berechnet werden. Die Umsetzung dieser Berechnung findet durch das Prinzip der Dynamischen Programmierung statt. Der Grundgedanke der Dynamischen Programmierung ist das Prinzip „Teile und Herrsche" und besagt, dass ein Problem erst in Teilprobleme unterteilt wird, diese dann gelöst werden und für die Lösung des Gesamtproblems die Teillösungen ausgenutzt werden (vgl. Sedgewick 2005, S. 673; Ottmann u. Widmayer 2012, S. 456). Algorithmus 9 beschreibt eine eigene Java-Implementierung für das Berechnen der Anzahl an Pfaden.

```
public static int getNumberOfPaths(Graph graph, String source, String sink) {
  int nrpaths = 0;

  //Topology order of all nodes.
  ArrayList<String> nodes = getTopologyOrder(graph);

  HashMap<String, Integer> values = new HashMap<String, Integer>();

  //All nodes with inDegree == 0 get value 1.
  for (String s : graph.getNodes()) {
    if ( (graph.inDegreeOf(s) == 0) ) {
      values.put(s, 1);
    }
  }

  //Iterate over all nodes and calculate their values.
  for (String v : nodes) {
    if (!values.containsKey(v)) {
      int sum = 0;
      for (Edge e : graph.getEdges()) {
        if (e.getSink().equals(v)) {
          String w = e.getSource();

          //Get value of already calculated nodes
          sum += values.get(w);
        }
      }
      //Write solution for current node
      values.put(s, sum);
    }
  }

  //Returns the value for the last node (sink).
  nrpaths = values.get(sink);
  return nrpaths;
}
```

Algorithmus 9: Java-Implementierung für die Berechnung der Pfadanzahl eines DAGs

Der erste Schritt ist das topologische Sortieren der Knotenmenge. Zu jedem Knoten wird dann in einer Hashmap ein zugehöriger numerischer Wert gespeichert, der die Anzahl an Pfaden zu diesem Knoten widerspiegelt. Die Knoten, die einen Eingangsgrad von 0 besitzen, erhalten den Initialisierungswert 1. Daraufhin werden die sortierten Knoten iterativ durchlaufen und für jeden Knoten v die Kanten ermittelt, deren Endknoten diesem Knoten v entsprechen. Die bereits gespeicherten Werte der Startknoten der zutreffenden Kanten werden aufsummiert und als Wert für den aktuellen Knoten v in der Hashmap gespeichert. Der Wert für den letzten durchlaufenen Knoten v entspricht der Gesamtzahl an Pfaden von der Quelle zur Senke des gegebenen DAGs und besteht somit aus den Lösungen seiner Teilprobleme.

Abbildung 5.5 zeigt eine Anwendung dieses Algorithmus. Zur besseren Lesbarkeit werden die Kantenkapazitäten nicht angezeigt. Die roten Zahlen zeigen die numerischen Werte, die für jeden Knoten in der

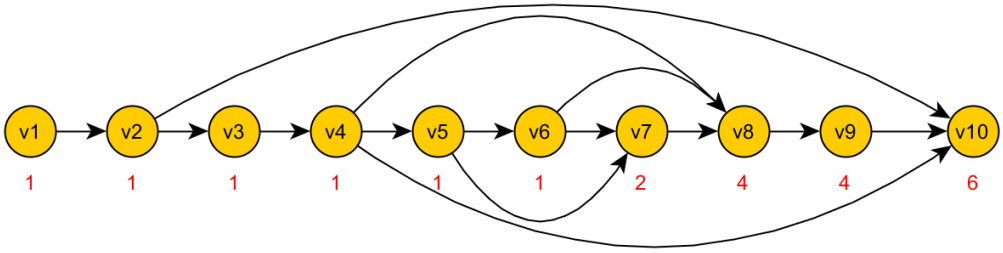

Abb. 5.5: Visualisierung für die Berechnung der Pfadanzahl in einem DAG

Hashmap gespeichert werden. Der Wert des letzten Knotens, hier Knoten v_{10}, beträgt 6 und ist die Gesamtanzahl an Pfaden von der Quelle zur Senke.

Die Ausgabe der Hashmap während der einzelnen Iterationen des Algorithmus kann in Tabelle 5.2 betrachtet werden. Anhand der Ausgabe kann dargelegt werden, dass jeder neue berechnete Wert auf den vorher berechneten Werten beruht:

Iteration	**Werte in Hashmap**
1	v1=1
2	v1=1, v2=1
3	v1=1, v2=1, v3=1
4	v1=1, v2=1, v3=1, v4=1
5	v1=1, v2=1, v3=1, v4=1, v5=1
6	v1=1, v2=1, v3=1, v4=1, v5=1, v6=1
7	v1=1, v2=1, v3=1, v4=1, v5=1, v6=1, v7=2
8	v1=1, v2=1, v3=1, v4=1, v5=1, v6=1, v7=2, v8=4
9	v1=1, v2=1, v3=1, v4=1, v5=1, v6=1, v7=2, v8=4, v9=4
10	v1=1, v2=1, v3=1, v4=1, v5=1, v6=1, v7=2, v8=4, v9=4, v10=6

Tab. 5.2: Ergebnisse der Iterationen der Berechnung der Pfadanzahl

Anhang E - Parametrisierung der GlpK-Bibliothek

Die folgenden Abschnitte beschreiben die Untersuchungsergebnisse der drei GlpK-Parameter: Branching-Algorithmus, Backtracking-Algorithmus und Schnittebenenverfahren. Dabei werden jeweils in 10 Durchläufen ein randomisiertes Flussnetz und eine dazugehörige Pfadmenge generiert und diese Probleminstanz mit dem jeweiligen eingestellten Parameter berechnet. Die Messzeiten werden in Millisekunden angegeben. Das arithmetische Mittel der Messzeit zeigt den Durchschnittswert für die jeweilige Einstellung. Da die Messzeiten bei einer gleichbleibenden Problemgröße stark variieren, wird zusätzlich der Median berechnet, da dieser weniger anfällig für ausreißende Werte ist. Das Verfahren, welches bei der Berechnung einer Probleminstanz als bestes abgeschnitten hat, wird grün, das schlechteste rot markiert. Das Ziel ist es, einen Trend zu erkennen, ob ein Parameter für die gegebene Fragestellung des Path-Packings die vorliegenden Strukturen der Gleichungssysteme ausnutzen kann und so eine verbesserte Laufzeit resultiert.

Auswertung der Branching-Parameter

n	FFV	LFV	MFV	DTH (default)	PCH
n=300	859	1.735	654	933	1.938
	412	389	441	1.408	1.250
	47.417	14.071	16.485	45.938	24.965
	2.894	2.588	1.125	1.956	2.970
	1.717	2.644	835	1.532	1.544
	17.143	1.399	5.089	3.491	3.935
	171	937	166	195	475
	216	276	283	290	640
	370	365	600	1.246	1.219
	365	649	569	3.538	1.610
Mittelwert	**7.156**	**2.505**	**2.625**	**6.053**	**4.055**
Median	**636**	**1.168**	**627**	**1.470**	**1.577**
n=400	30.325	21.040	12.086	134.998	16.030
	2.966	3.131	2.999	28.322	15.620
	7.669	5.234	2.359	18.221	8.164
	48.996	18.750	30.065	43.795	11.221
	7.257	3.416	8.264	21.979	13.889
	5.220	1.679	2.451	5.023	5.920
	8.064	6.771	4.035	11.737	8.719
	5.666	3.654	2.388	3.649	2.975
	4.196	3.416	1.995	4.309	4.189
	7.120	4.429	8.296	9.932	9.628
Mittelwert	**12.748**	**7.152**	**7.494**	**28.197**	**9.636**
Median	**7.189**	**4.042**	**3.517**	**14.979**	**9.174**
n=500	32.394	43.262	26.865	60.935	15.567
	6.577	30.049	9.772	19.043	7.733
	146.698	125.487	74.510	308.842	125.683
	126.499	66.550	74.431	158.133	90.407
	45.550	46.322	14.420	56.375	15.362
	42.488	26.718	12.473	121.949	79.074
	3.764	1.611	1.163	7.764	4.994
	54.470	1.425.348	201.690	597.308	150.832
	99.864	143.350	100.954	156.168	120.613
	110.605	103.964	47.104	205.591	187.142
Mittelwert	**66.891**	**201.266**	**56.338**	**169.211**	**79.741**
Median	**50.010**	**56.436**	**36.985**	**139.059**	**84.741**

Tab. 5.3: Auswertung der Branching-Parameter

Tabelle 5.3 zeigt die Messzeiten der Branching-Parameter, die für kleine Problemgrößen berechnet worden sind. Der eingestellte Standardwert der GlpK-Bibliothek für den Branching-Algorithmus ist die Heuristik nach Driebeck und Tomlin. Die Implementierung dieser Heuristik der GlpK-Bibliothek basiert auf (Driebeck 1966; Tomlin 1970). Es ist erkennbar, dass diese Heuristik für die eingestellten Problemgrößen $n = 300, 400$ und 500 für fast alle erzeugten Probleminstanzen am schlechtesten abschneidet. Besonders gut hingegen schnitt das MFV-Verfahren ab, welches für das Branching die meist gebrochene Variable wählt. In anderen Worten: es wählt die Variable, die am nächsten an der Mitte zweier Ganzzahlen liegt (GnuProject 2014e, Z. 134ff.). Alle anderen Verfahren zeigen ein mittelmäßiges Verhalten. Auf Grundlage der beschriebenen Ergebnisse wird der Parameter für das Branching von der Heuristik nach Driebeck und Tomlin zum MFV-Verfahren abgeändert.

Auswertung der Backtracking-Parameter

n	DFS	BFS	BLB(default)	BPH
n=300	1.680	575	630	1.411
	10.120	6.586	2.956	4.089
	249	246	251	246
	2.882	708	509	506
	945	977	626	741
	599	384	330	377
	210	210	209	209
	881	636	592	539
	95.332	92.981	57.316	80.350
	5.129	5.147	3.635	2.883
Mittelwert	**11.803**	**10.845**	**6.705**	**9.135**
Median	**1.313**	**672**	**609**	**640**
n=400	323.475	280.409	108.693	72.969
	443.332	34.780	10.349	81.251
	23.128	10.377	7.003	8.157
	12.736	7.220	5.986	6.524
	481.896	105.892	62.377	34.035
	66.342	17.910	15.067	17.085
	1.727	2.261	1.645	1.609
	173.283	81.185	88.751	82.990
	84.475	39.209	74.445	25.911
	543.894	661.703	216.885	192.859
Mittelwert	**215.429**	**124.095**	**59.120**	**52.339**
Median	**128.879**	**36.995**	**38.722**	**29.973**
n=500	2.834.748	2.801.969	2.237.425	525.311
	2.734.202	1.158.700	443.561	344.929
	5.492.667	5.232.054	4.163.388	3.219.994
	333.589	177.797	140.007	92.596
	454.464	343.011	820.470	373.168
	112.730	18.781	8.437	10.446
	221.988	188.646	77.437	131.813
	564.963	329.061	326.551	240.275
	365.325	116.217	215.554	140.321
	187.684	117.592	57.936	85.684
Mittelwert	**1.330.236**	**1.048.383**	**849.077**	**516.454**
Median	**409.895**	**258.854**	**271.053**	**190.298**

Tab. 5.4: Auswertung der Backtracking-Parameter

Die Messzeiten der Backtracking-Verfahren (vgl. Tab. 5.4) zeigen eindeutig, dass das DFS-Verfahren am schlechtesten abschneidet. Die Verfahren BLB und BPH zeigen das beste Verhalten. Die Entscheidung für die Wahl der Einstellung fällt allerdings auf das BPH-Verfahren, da dies vor allem für die Problemgröße $n = 500$ im Vergleich zu allen anderen Verfahren am besten abgeschnitten hat. Die Hoffnung ist, dass für noch größere Probleminstanzen dieses Verfahren ebenfalls gegenüber dem BLB-Verfahren dominiert. Das Standardverfahren BLB wird daher ersetzt durch das BPH-Verfahren.

Auswertung der Schnittebenenverfahren

n	None(def.)	GMI	MIR	COV	CLQ
n=300	4.794	24.625	5.462	4.946	4.833
	225	222	207	207	197
	355	539	360	343	329
	553	1.007	620	538	528
	2.696	1.881	2.866	2.680	2.622
	224	197	208	204	199
	6.375	15.569	7.289	6.547	6.307
	1.179	1.636	1.164	1.187	1.132
	398	3.106	3.106	359	374
	539	594	529	512	503
Mittelwert	**1.734**	**4.938**	**2.181**	**1.752**	**1.702**
Median	**546**	**1.322**	**892**	**525**	**516**
n=400	616.660	641.583	604.366	622.041	609.477
	36.123	18.916	36.941	36.878	35.453
	78.956	110.795	84.548	81.931	77.875
	654.168	628.066	706.424	689.783	673.687
	372.956	612.130	363.919	389.178	375.998
	3.722	5.491	3.888	3.874	3.768
	17.704	34.508	19.267	18.249	17.833
	5.680	7.053	6.059	5.904	5.616
	11.601	19.785	12.647	12.091	11.570
	239.759	531.043	226.360	224.050	219.970
Mittelwert	**203.733**	**260.937**	**206.442**	**208.398**	**203.125**
Median	**57.540**	**72.652**	**60.745**	**59.405**	**56.664**
n=500	45.872	32.205	48.897	47.384	45.658
	33.507	18.001	34.601	33.816	33.839
	94.319	19.513	90.786	89.368	88.254
	592.939	903.872	642.091	616.976	599.873
	422.798	597.259	441.452	433.074	417.435
	38.919	77.649	41.230	39.614	38.893
	563.575	1.535.825	611.920	588.135	566.572
	530.048	2.151.896	550.174	534.160	526.651
	156.253	104.586	164.798	158.940	157.309
	279.752	536.355	287.463	277.568	271.582
Mittelwert	**275.798**	**597.716**	**291.341**	**281.904**	**274.607**
Median	**218.003**	**320.471**	**226.131**	**218.254**	**214.446**

Tab. 5.5: Auswertung der Schnittebenenverfahren

Laut der Dokumentation der GlpK-Bibliothek (GnuProject 2014d, S. 61ff.) verwendet diese das Branch & Cut-Verfahren. Allerdings sind alle Schnittebenenverfahren standardmäßig deaktiviert, sodass eigentlich ein Branch & Bound-Verfahren resultiert. Die angebotenen Schnittebenenverfahren werden auf Basis der genannten Problemgrößen untersucht, um ein Verfahren ausfindig zu machen, welches die Berechnung für das Path-Packing beschleunigen kann. Tabelle 5.5 legt die Messzeiten der Schnittebenenverfahren dar.

Das GMI-Verfahren zeigt ein sehr uneinheitliches Verhalten. In fast allen Testfällen zeigt es mit Abstand das schlechteste Verhalten; in einigen anderen Fällen dominiert es aber im Vergleich zu den anderen Verfahren. Ersichtlich ist ebenfalls, dass ein aktives Schnittebenenverfahren zusätzliche Laufzeit braucht. Daher ist es auch nicht verwunderlich, dass der Standardfall, nämlich die Deaktivierung aller Schnittebenenverfahren, in einigen Fällen als „Gewinner" hervorgeht. Das positivste Verhalten zeigt jedoch das CLQ-Schnittebenenverfahren. Die Implementierung des CLQ-Verfahren (GnuProject 2014b, Z. 30ff.) beruht auf einem Konfliktgraphen, der angibt, ob zwei Variablen gleichzeitig mit dem Wert 1 belegt werden können. Für das Path-Packing bedeutet dies, dass zwei Pfade, die mit den Variablen x_i und x_j repräsentiert werden, in Konflikt miteinander stehen. Die Schnittebene wird nun auf Basis der größten gewichteten Clique dieses Konfliktgraphen gebildet. Dabei beruht das Finden dieser größten Clique auf dem Algorithmus von Östergard (Östergard 2002; GnuProject 2014h).

Ein ebenfalls gutes Verhalten zeigt das COV-Schnittebenenverfahren. Die Implementierung des COV-Verfahrens (GnuProject 2014c, Z. 28ff.) zeigt die enge Verwandtschaft mit dem Knapsack-Problem. Die Gemeinsamkeiten des Knapsack-Problems mit dem Path-Packing ist in dieser Arbeit schon in Kapitel 2.5.1 gezeigt worden. Die Messergebnisse dieses Verfahrens decken daher die theoretischen Vergleiche dieser Probleme.

Da die Messzeiten für das CLQ-Verfahren etwas besser sind als die des COV-Verfahren, fällt die Wahl auf das CLQ-Verfahren.

Anhang F - Inhalt der beiliegenden DVD

Das beiliegende Medium beinhaltet alle Inhalte, die für die Anfertigung dieser Abschlussarbeit relevant sind. Folgende Ordnerstruktur existiert auf diesem Medium :

- *Arbeit* : Beinhaltet die Abschlussarbeit als pdf-Dokument.
- *Ergebnisse* : Dieser Ordner beinhaltet alle gemessenen Daten, unter anderem die Messzeiten der einfachen und schweren Instanzen sowie die Messzeiten für die Analyse der drei untersuchten GlpK-Parameter.
- *Exposee* : Beinhaltet das eingereichte Exposee der Abschlussarbeit.
- *Grafiken* : Alle erstellten Abbildungen, wie bspw. die Flussnetze oder auch das UML-Diagramm befinden sich in diesem Ordner.
- *Quellen* : Eine Kopie der verwendeten Versionen der angebundenen Bibliotheken werden auf diesem Medium mitgeliefert.
- *Workspace* : Die selbst entwickelte Path-Packing- und ConsoleTable-Bibliothek befinden sich in diesem Ordner.